Elementi di Algoritmi Geometrici

A. Fusiello

- *Introduzione*
- *Preliminari*
- *Triangolazione del poligono*
- *Guscio convesso*
- *Intersezioni*
- *Suddivisioni piane*
- *Ricerca geometrica*
- *Prossimità*

Introduzione

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

> La geometria computazionale studia gli ***algoritmi*** e le ***strutture dati*** atti a risolvere in modo efficiente problemi di natura ***geometrica***.

- L'efficienza si riferisce all'impiego di ***risorse*** da parte della computazione, in termini di tempo impiegato e spazio occupato. Si parla anche di ***complessità*** dell'algoritmo.
- 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*** (rimozione superfici nascoste, collisioni), ***Robotica*** (visibilità e pianificazione del moto), ***CAD*** (progetto di circuiti integrati) e ***Sistemi Informativi Territoriali*** (GIS) (localizzazione, ricerca, pianificazione).

- La nozione di ***algoritmo*** in geometria è molto antica: si può fare risalire alla ***costruzione Euclidea***, che definisce una collezione di strumenti consentiti (riga e compasso) ed una serie di operazioni che è consentito compiere con essi. In questo contesto una dimostrazione o una costruzione definiscono quindi implicitamente un algoritmo.
- La nozione di ***complessità*** in geometria è più recente, ma comunque precede l'avvento del calcolatore. Nel 1902 E. Lemoine codifica le operazioni Euclidee primitive e definisce la "semplicità" di una costruzione geometrica come il numero di operazioni effettuate.
- La Geometria Computazionale (GC), nella sua accezione prevalente, nasce nel 1975 con una pubblicazione di M. I. Shamos.
- All'inizio la GC si è concentrata sulla generalizzazione multi-dimensionale (soprattutto 2-d) di problemi 1-d, come ordinamento e ricerca.
- Gli aspetti legati alla natura discreta dei problemi geometrici hanno ricevuto maggiore attenzione, rispetto quelli di natura continua.
- Gli oggetti di interesse sono "rettilinei" (rette, segmenti, poligoni, semipiani) o curve molto semplici (cerchio).

Preliminari

Geometrici

- Gli oggetti considerati in Geometria Computazionale sono solitamente insiemi di punti nello ***spazio Euclideo*** E^d.
- Gli oggetti geometrici devono essere ***finitamente specificabili***, ovvero non necessariamente costituiti da un numero finito di punti, ma (p. es.) specificabili con un numero finito di parametri.
- I ***punti:*** sono denotati da lettere stampatello maiuscole, come $P, Q, R, \ldots$.
- Gli ***insiemi*** di punti sono denotati da lettere corsivo maiuscole, come $\mathcal{O}, \mathcal{V}, \mathcal{S}, \ldots$
- ***Aperti:*** Un insieme $\mathcal{S}$ di E^d si dice aperto se ogni punto vi appartiene con tutto un intorno. La ***frontiera*** di $\mathcal{S}$, denotata da $\partial\mathcal{S}$, è l'insieme dei punti tali che ciascun intorno contiene sia punti di $\mathcal{S}$ che del suo complementare. Un insieme ***chiuso*** contiene tutti i suoi punti di frontiera, un insieme ***aperto*** non ne contiene nessuno. L'interno di $\mathcal{S}$ è $\mathcal{S} \setminus \partial\mathcal{S}$, ovvero il più grande aperto contenuto nell'insieme stesso.
- Si assume che sia assegnato un ***sistema di riferimento cartesiano***, per cui i

punti sono rappresentati da n-ple di coordinate. P.es. nel piano E^2 si ha: $P = (P_x, P_y)$.

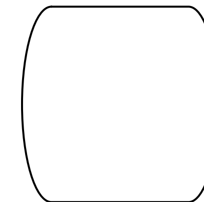
- Dati due punti P_0 e P_1 e dati due scalari α_1 e α_2 t.c. $\alpha_1 + \alpha_2 = 1$ definiamo la ***combinazione affine*** di P_0 e P_1 come:

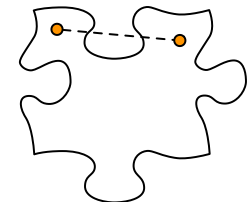
$$\alpha_1 P_1 + \alpha_2 P_2 = P_0 + \alpha_1 (P_1 - P_0)$$

- La combinazione affine di due punti distinti descrive la ***retta*** passante per i due punti.

- La ***combinazione convessa*** di P_0 e P_1 si ottiene imponendo che $\alpha_1, \alpha_2 \geq 0$. La combinazione convessa di P_0 e P_1 descrive il ***segmento*** $\overline{P_o P_1}$.

- Un insieme $\mathcal{C}$ in E^d è ***convesso*** se per ogni coppia di punti P_1 e P_2 appartenenti a $\mathcal{C}$ si ha che $P' = \alpha(P_1 - P_2) + P_2$ appartiene a $\mathcal{C}$ per ogni $\alpha \in [0, 1]$ ovvero tutti i punti sul segmento che unisce P_1 con P_2 appartengono all'insieme $\mathcal{C}$.


Convesso


Non convesso

- Si vede facilmente che l'intersezione di insiemi convessi è convessa.

- A volte sarà utile considerare rette e segmenti ***orientati*** (p.es. la retta scritta sopra è orientata da P_0 a P_1). Indicheremo con $\overrightarrow{PQ}$ il segmento orientato da P a Q. Con abuso di

notazione indicheremo allo stesso modo la retta orientata che contiene il segmento (il contesto servirà a disambiguare).

- In E^d una equazione lineare come

$$a_1 P_1 + a_2 P_2 + \ldots a_d P_d = c$$

definisce un ***iperpiano***. Il vettore $a_1, \ldots, a_d$ è ortogonale all'iperpiano.

- L'insieme di punti che giacciono da una parte dell'iperpiano (soddisfano la formula qui sopra con $>$ o con $<$) si chiama ***semispazio***. L'iperpiano si dice ***di supporto*** per il semispazio.

- In E^2 gli ***iperpiani sono le rette***. L'equazione sopra si specializza in:

$$ax + by = c.$$

Un'altra forma, che non rappresenta però le rette verticali è:

$$y = ax + b.$$

- In E^2 i semispazi si chiamano ***semipiani***. Se consideriamo la retta orientata, il semipiano che giace alla destra/sinistra della retta prende il nome di semipiano destro/sinistro.

Algoritmici

Analisi degli algoritmi

- Analizzare un algoritmo vuol dire stimarne la ***complessità*** o ***costo computazionale***, ovvero la quantità di ***risorse*** che esso richiede per l'esecuzione: tempo e spazio (soprattutto tempo).
- Per fare questo occorre un ***modello dell'esecutore***: per rimanere indipendenti dalla particolare macchina si usa un modello generico, il classico modello RAM (***Random Access Machine***), nel quale le istruzioni sono eseguite una alla volta senza parallelismo e le celle di memoria contengono numeri reali.
- Non si contano le singole operazioni, ma ci si accontenta di stimare il tempo di esecuzione ***modulo una costante moltiplicativa***. Questo vuol dire che è lecito contare solo certe ***operazioni "chiave"*** (di solito confronti), supponendo che rappresentino una frazione costante di tutte le operazioni eseguite.
- Tipicamente il tempo di esecuzione cresce con il crescere della ***dimensione dell'input*** (p.es. il numero di elementi da ordinare).
- Quindi descriveremo il tempo di esecuzione dell'algoritmo come una ***funzione*** $T(n)$ della dimensione dell'input n.

- Inoltre il tempo di esecuzione dipende anche da come l'input si presenta (p.es. nell'ordinamento, il vettore di input può essere già ordinato, può essere ordinato inversamente, oppure a caso).

- Per essere indipendenti da questo aspetto, si considera il tempo di calcolo nel ***caso peggiore***. Questo perché:

  1. il tempo di calcolo nel caso peggiore rappresenta comunque un limite superiore al tempo di calcolo per ogni input e
  2. l'alternativa di considerare il ***caso medio*** non è sempre praticabile, poiché non sempre c'è accordo su cosa debba costituire il caso medio (e l'analisi è più complessa).

- Inoltre, non si è interessati alla esatta forma analitica di $T(n)$ quanto piuttosto al suo comportamento ***asintotico***, ovvero a delimitare $T(n)$ superiormente al crescere di n.

- Questo vuol dire (p.es.) che se $T(n)$ è un polinomio domina il termine di grado maggiore.

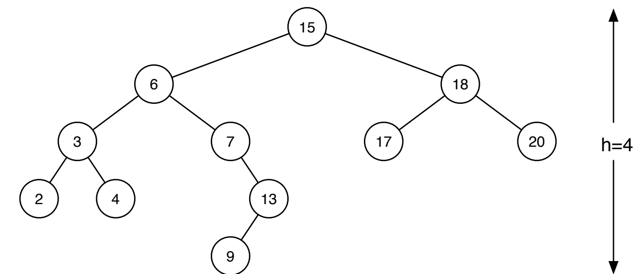
- Tutto ciò è reso formalmente dalla seguente notazione (Knuth, 1976):
  Diremo che $T(n) \in O(f(n))$ se

$$\lim_{n \to \infty} \frac{T(n)}{F(n)} = \mathrm{cost}$$

- In pratica se $T(n)$ è $O(f(n))$ vuol dire che $f(n)$ controlla la crescita asintotica di $T(n)$.

Alberi binari

- Un *albero binario* è una struttura definita su di un insieme di nodi tale che:
  - contiene zero nodi (albero binario vuoto), oppure
  - è composta da tre insiemi disgiunti di nodi: il nodo *radice*, un albero binario chiamato *sottoalbero destro*, ed un albero binario chiamato *sottoalbero sinistro*.



Un albero binario di ricerca

- Dato un nodo v, se il sottoalbero destro/sinistro è non vuoto, la sua radice si chiama *figlio* destro/sinistro.
- Un nodo senza figli si chiama *foglia* o nodo esterno. Gli altri si chiamano *nodi interni*.
- La *profondità* di un nodo è il numero dei suoi *antenati* (fino alla radice).
- La massima profondità dei nodi si chiama *altezza* dell'albero.
- Un albero binario in cui tutte le foglie hanno la stessa profondità si dice *completo*.
- In un albero binario completo, se l'altezza è h, il numero di foglie è 2^h. Il numero di

nodi interni è $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ (serie geometrica). Di conseguenza, l'altezza di un albero binario completo con n foglie è $\log_2 n$.

- Un ***albero binario di ricerca*** è una struttura dati nella quale i dati sono organizzati in un albero binario.
- Il dato, rappresentato da una ***chiave***, è archiviato nei nodi, in modo da soddisfare la seguente ***proprietà***: il sottoalbero sinistro di un nodo v contiene le chiavi minori di quella di v, il sottoalbero destro di v contiene le chiavi maggiori di quella di v.
- Si assume che le chiavi appartengano ad un insieme totalmente ordinato.
- Gli alberi binari di ricerca supportano molte operazioni del tipo di dato ***insieme***, ovvero (sia `S` l'insieme e `x` la chiave):
  - `Search(S,x)`
  - `Delete(S,x)`
  - `Minimum(S)`
  - `Maximum(S)`
  - `Successor(S,x)`
  - `Predecessor(S,x)`

- Tutte le operazioni elencate hanno un costo computazionale proporzionale all'altezza dell'albero. Per un albero con n nodi l'altezza può essere $O(\log_2 n)$ ma anche $O(n)$ nel caso peggiore.

- I ***red-black trees*** sono alberi binari di ricerca che mantengono una proprietà di ***bilanciamento***, ovvero evitano che nessun nodo abbia profondità doppia di quella di qualcun'altro. In questo modo l'altezza è garantita $O(\log_2 n)$ e le operazioni sopra elencate costano $O(\log_2 n)$.

- A titolo di esempio, vediamo lo pseudocodice di `Search`, che accetta come argomenti un puntatore al nodo radice `t` ed il valore da ricercare `x`.

```
Search(t, x)
if  ( t = NIL or x = key(t) )
   then return t
if x < key(t)
   then return Search(left(t), x)
   else return Search(right(t), x)
```

- ***Complessità***. Il tempo di calcolo è proporzionale al numero di nodi visitati. Nel caso peggiore il valore cercato è in una foglia, quindi $O(h)$ dove h è l'altezza.

Il paradigma "divide-et-impera" e MergeSort

- Il paradigma "divide-et-impera" (*divide and conquer*) comprende tre fasi:

  Dividi il problema in un certo numero (tip. 2) di sottoproblemi.

  Domina i sottoproblemi, risolvendoli ricorsivamente.

  Combina le soluzioni dei sottoproblemi in una soluzione del problema originale

- Il ***MergeSort*** (algoritmo per l'ordinamento di vettori) segue da vicino il paradigma. Esso procede nel seguente modo:

  Dividi: partiziona il vettore di n elementi in due vettori di $n/2$ elementi ciascuno;

  Domina: ordina i due sotto-vettori ricorsivamente usando MergeSort;

  Combina: immergi i due vettori ordinati in un solo vettore ordinato.

- L'operazione chiave è il passo "combina" (*Merge*), che richiede del lavoro (gli altri due sono banali). Si vede facilmente che l'operazione può essere compiuta in tempo lineare nella somma delle lunghezze dei due vettori da combinare (infatti si fa in una passata).

- ***Complessità***. Il costo computazionale di MergeSort è la soluzione della seguente ricorrenza :

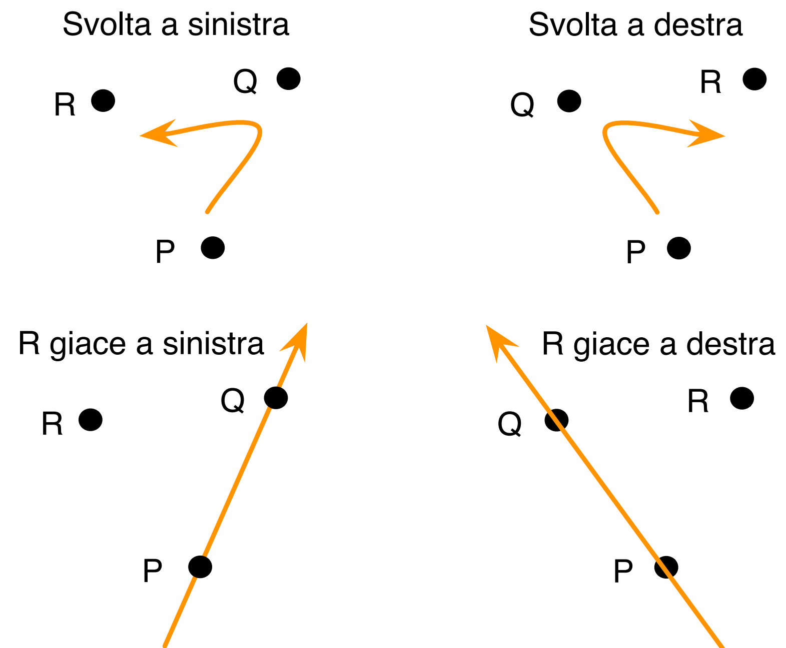
$$T(n) = \begin{cases} 1 & \text{se } n = 2 \\ 2T(n/2) + n & \text{altrimenti} \end{cases}$$

Si vede con il metodo dell'iterazione che la soluzione è $T(n) = n \log n$. Infatti, sostituendo ripetutamente l'equazione in se stessa si ottiene, al passo i-esimo: $T(n) = 2^i T(\frac{n}{2^i}) + in$. Assumiamo che n sia potenza di due, ovvero $n = 2^{i+1}$. Dunque: $T(n) = nT(2) + (\log_2 n - 1)n = n \log n$.

Operazioni di base

Orientamento di una terna di punti

- L'orientamento di tre punti è l'analogo geometrico del confronto tra reali.
- Come il confronto, servirà spesso a prendere decisioni negli algoritmi che vedremo.
- L'***orientamento*** di tre punti del piano $\text{orient}(P, Q, R)$ stabilisce se la terna P, Q, R definisce una svolta a destra (orario) o a sinistra (antiorario).
- Equivalentemente, $\text{orient}(P, Q, R)$ stabilisce da che parte si trova R rispetto a $\overrightarrow{PQ}$, la retta passante per P e Q, orientata da P a Q.



- La retta $\overrightarrow{PQ}$ ha equazione:

$$(x - P_x)(Q_y - P_y) - (y - P_y)(Q_x - P_x) = 0$$

Sostituendo le coordinate di R a (x, y) si ottiene uno scalare a, il cui segno determina la posizione di R. In particolare:

  - se $a > 0$, R giace a *sinistra* di $\overrightarrow{PQ}$ e la terna definisce svolta a sinistra (antiorario);
  - se $a < 0$, R giace a *destra* di $\overrightarrow{PQ}$ e la terna definisce svolta a destra (orario);
  - se $a = 0$, i tre punti sono allineati.

- Osserviamo che a è il determinante della matrice:

$$\begin{bmatrix} R_x - P_x & Q_x - P_x \\ R_y - P_y & Q_y - P_y \end{bmatrix}$$

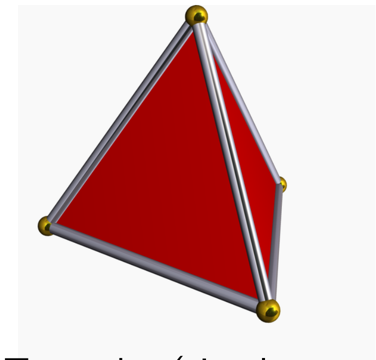
- Quindi definiamo formalmente l'orientamento:

$$\text{orient}(P, Q, R) = \text{sign}(a)$$

- ***Area del triangolo***: si dimostra che $|a|$ è pari al doppio dell'area del triangolo che ha per vertici P, Q, R.
- Si vede facilmente che la seguente espressione per a è equivalente:

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

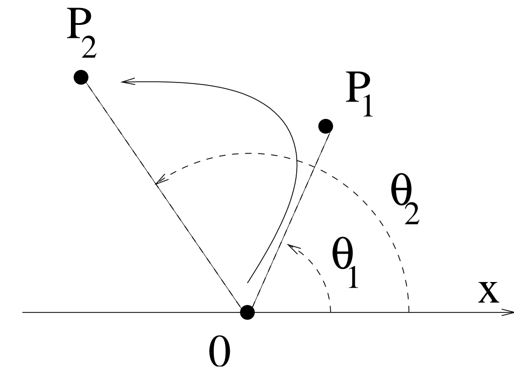
- Questa formula per a si generalizza a n dimensioni.
- Per esempio, in 3D, $\text{orient}(P, Q, R, S)$ stabilisce da che parte si trova S rispetto al piano orientato individuato da tre punti PQR. Il valore assoluto di a è pari a 6 volte il volume del tetraedro che ha per vertici P, Q, R, S.


Tetraedro (simplesso 3D).

- In 1D, $\text{orient}(P, Q) = \text{sign}(P - Q)$ si riduce al confronto $P < Q$, quindi, in un certo senso, $\text{orient}()$ generalizza l'operazione di confronto.

- Un altro modo di vedere che orient() generalizza l'operazione di confronto è il seguente.

- ***Confronto polare***: dato un sistema di riferimento polare, formato dall'asse x e da un polo O, 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 O, P_1, P_2 è una terna antioraria (ovvero $\text{orient}(O, P_1, P_2) = 1$)

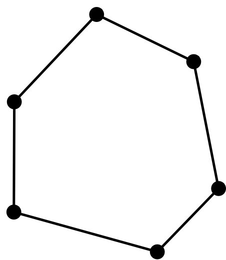
- ***Complessità***: la complessità del calcolo di orient() nel piano è costante ($O(1)$) poiché il numero di operazioni è fissato.

- Si noti che queste operazioni di base non usano divisioni nè funzioni trigonometriche, che sono affette da errori di arrotondamento (ed anche computazionalmente onerose).

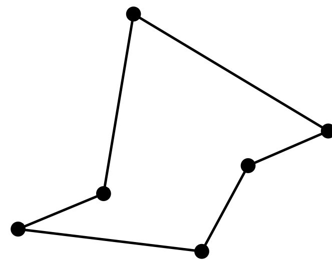
Triangolazione del poligono

Poligoni

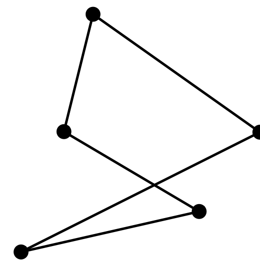
- Informalmente, un ***poligono*** è una linea spezzata del piano che si chiude su se stessa.
- Formalmente: Un poligono $\mathcal{P}$ è un insieme finito di segmenti (spigoli) di E^2, in cui ogni estremo (vertice) è comune a esattamente due segmenti, che si dicono *adiacenti*.
- Oppure: un poligono $\mathcal{P}$ è costituito da una una successione ordinata e finita di punti $P_1 \dots P_n$ di E^2 (vertici) e dai segmenti $\overline{P_1P_2} \dots \overline{P_nP_1}$ (spigoli) con la proprietà che due spigoli consecutivi (adiacenti) si intersecano solo in corrispondenza del vertice comune.



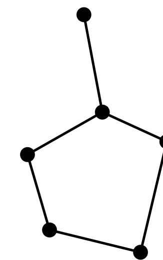
Poligono, convesso



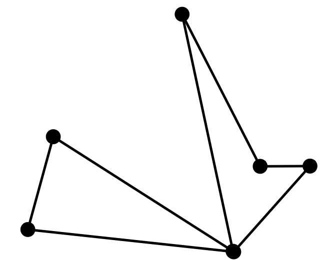
Poligono, non convesso



Poligono, non semplice



Non poligono



Non poligono

- Un poligono è detto ***semplice*** se ogni coppia di spigoli non adiacenti ha intersezione vuota.

- ***Teorema di Jordan***: Un ***poligono semplice*** $\mathcal{P}$ divide il piano in due regioni o facce, una limitata (detta ***interno*** di $\mathcal{P}$) ed una illimitata (detta ***esterno*** di $\mathcal{P}$).

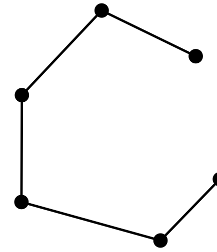
  (Nota: Spesso si usa il termine poligono anche per riferirsi alla regione del piano definita dal poligono semplice unito al suo interno.)

- Non ci occuperemo di poligoni non semplici.

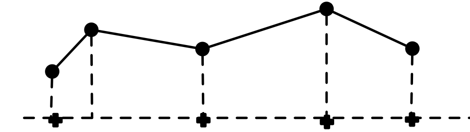
- Un poligono semplice è ***convesso*** se il suo interno è convesso.

- Per convenzione, un poligono viene rappresentato dalla sequenza dei suoi vertici $P_1 \ldots P_n$ ordinati in modo che l'interno del poligono giaccia alla sinistra della retta orientata da P_i a P_{i+1}, ovvero i vertici sono ordinati in senso ***antiorario***.

- Definizione: Se nella definizione di poligono rimuoviamo l'ultimo segmento $\overline{P_n P_1}$ otteniamo una ***catena poligonale***.



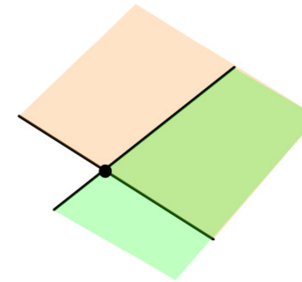
Catena poligonale



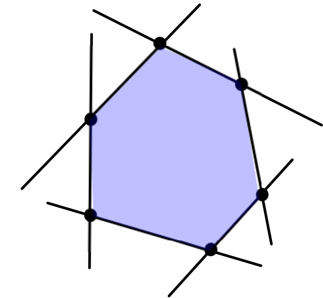
Catena poligonale monotona

- Una catena poligonale $\mathcal{C}$ si dice monotona rispetto alla retta ℓ se una retta ortogonale ad ℓ interseca $\mathcal{C}$ in un solo punto.

- Un ***insieme poligonale*** è l'intersezione di un numero finito di semipiani chiusi.



Insieme poligonale

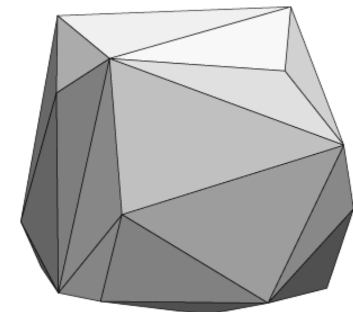


Insieme poligonale limitato

- L'insieme poligonale è ***convesso***, in quanto intersezione di insiemi convessi.

- L'insieme poligonale può essere limitato o illimitato. Nel primo caso è un poligono convesso.

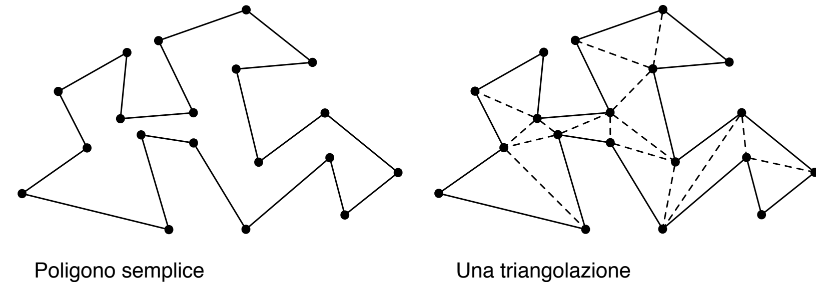
- ***Poliedri*** In E^3 un poliedro semplice è definito da una insieme finito di poligoni (facce) tali che ciascuno spigolo di una faccia è condiviso da esattamente un'altra faccia e le facce non si intersecano che negli spigoli.



Triangolazione di poligoni semplici

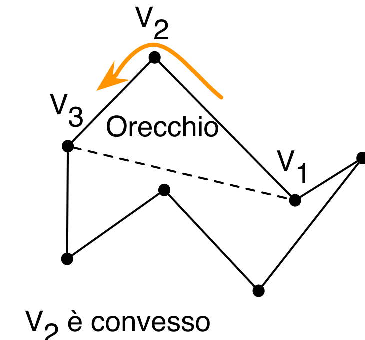
- ***Diagonale***: segmento che unisce due vertici del poligono t.c. il suo interno è contenuto nell'interno del poligono.

- ***Triangolazione di un poligono*** $\mathcal{P}$: insieme di triangoli i cui spigoli sono spigoli di $\mathcal{P}$ oppure diagonali, e la cui unione sortisce il poligono $\mathcal{P}$.



- La chiave per dimostrare l'esistenza della triangolazione è provare l'esistenza della diagonale. Prima alcune definizioni e semplici osservazioni.

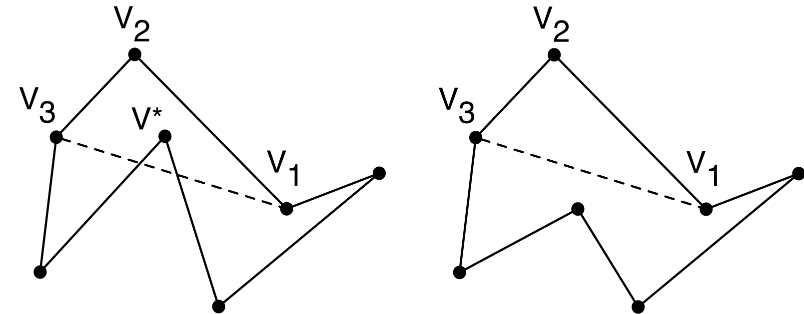
- Un vertice V_2 di un poligono si dice ***convesso*** se il suo angolo interno è minore di π.
  - È immediato verificare che V_2 è convesso se e solo se assieme al suo predecessore V_1 ed al suo successore V_3 (percorrendo il poligono in senso antiorario) definiscono una ***svolta a sinistra***.
  - Inoltre, se V_2 è convesso e $\overline{V_1V_3}$ è una diagonale, il triangolo $V_1V_2V_3$ si chiama ***orecchio***.



- ***Oss.*** Ogni poligono ha almeno un vertice convesso. Basta prendere il vertice di ordinata minima (quello più a destra in caso di parità).

- ***Lemma*** (Meisters). Ogni poligono semplice $\mathcal{P}$ con $n \geq 4$ spigoli ha una diagonale.

- Dim. Sia V_i il vertice convesso. Se $\overline{V_{i-1}V_{i+1}}$ è contenuto in $\mathcal{P}$, allora è una diagonale (il triangolo $V_{i-1}V_iV_{i+1}$ è un orecchio). Altrimenti il triangolo $V_{i-1}V_iV_{i+1}$ contiene almeno un vertice di $\mathcal{P}$, per la convessità. Sia V^* il primo vertice che si incontra muovendo una retta parallela a $\overline{V_{i-1}V_{i+1}}$ da V_i verso $V_{i-1}V_{i+1}$. Il segmento $\overline{V_iV^*}$ è una diagonale.



- In particolare, si dimostra che ogni poligono con $n \geq 4$ spigoli ha almeno due ***orecchi*** (Two Ears Theorem, v. [3]).

- Questo sarà utile per trovare un algoritmo di triangolazione. Prima però stabiliamo l'esistenza della triangolazione stessa.

- ***Teorema***. Ogni poligono semplice di n vertici può essere triangolato (i), e la triangolazione consiste di $n-2$ triangoli e $n-3$ diagonali (ii).

- Dim. (i) Per induzione sul numero di vertici. Se $n=3$ è un triangolo. Se $n>3$ il poligono ammette una diagonale, che lo divide in due parti, ciascuna delle quali ha un numero di spigoli strettamente minore di n, e dunque è triangolabile per ipotesi induttiva. L'unione delle due triangolazioni è una triangolazione per il poligono.

- Dim. (ii) Per induzione sul numero di vertici.

- ***Problema:*** (Polygon Triangulation) Dato un poligono semplice, determinare le diagonali che lo suddividono in triangoli.

- Applicazione: grazie alla triangolazione si può calcolare ***l'area di un poligono semplice*** come somma dell'area dei triangoli (v. [3]).

- Oss: Un ***poligono convesso***, invece si triangola banalmente in $O(n)$ collegando un vertice con gli altri $n-3$ non adiacenti.

Algoritmo del taglio degli orecchi

- ***Risolve*** POLYGON TRIANGULATION
- ***Idea***: restringere il poligono tagliando via un orecchio alla volta fino a rimanere con un solo triangolo.
- Visita il contorno del poligono in senso antiorario.
- Per ogni terna di vertici consecutivi V_1, V_2, V_3 del poligono che svoltano a sinistra (candidati ad essere orecchio), controlla se il segmento $\overline{V_1V_3}$ è una diagonale:
  - In caso negativo, prosegui la ricerca spostando la terna V_1, V_2, V_3 in avanti di uno, ovvero controlla V_2, V_3, V_4, dove V_4 è il successore di V_3.
  - In caso positivo, V_1, V_2, V_3 formano un orecchio, quindi ritaglia l'orecchio dal poligono, aggiorna la triangolazione e procedi. Avendo appena cancellato V_2, la prossima terna da controllare è V_0, V_1, V_3 dove V_0 è il predecessore di V_1.
- Termina quando il poligono iniziale è ridotto ad un triangolo.

- Pseudocodice (adattato da [5]).

```
Sia P il poligono da triangolare.
T := empty set;
v1 := primo vertice di P;
while numero vertici in P > 3
  v2 := successore di v1 in P;
  v3 := successore di v2 in P;
  if (orient(v1,v2,v3) > 0) and
     (il segmento v1-v3 e' diagonale di P)
  then
     aggiungi la diagonale v1-v3 in T;
     rimuovi v2 da P;
     v1 := predecessore di v1 in P;
  else
     v1 := successore di v1 in P;
end /* while */
return T;
```

- Dettaglio: Per controllare se $\overline{V_1V_3}$ è una diagonale è sufficiente controllare che nessun vertice di $\mathcal{P}$ situato fra V_3 e V_1 (esclusi) sia interno al triangolo V_1, V_2, V_3.

- Un punto è interno al triangolo PQR (vertici dati in senso antiorario) sse giace alla sinistra di $\overrightarrow{PQ}$, $\overrightarrow{QR}$ e $\overrightarrow{RP}$.

- ***Complessità***. Per triangolare un poligono di n spigoli occorre tagliare $n-3$ orecchi. Il ciclo viene eseguito $n-3$ volte. Ad ogni ciclo il controllo della diagonale ha costo è lineare nel numero corrente di vertici di P, che sono $n-1$ al passo i-esimo. Dunque in totale il costo è $\sum_{i=1}^{n-3}(n-i) = O(n^2)$.

- Si può fare in $O(n \log n)$ con una tecnica che sfrutta un partizionamento in poligoni *monotoni* (v. [2]).

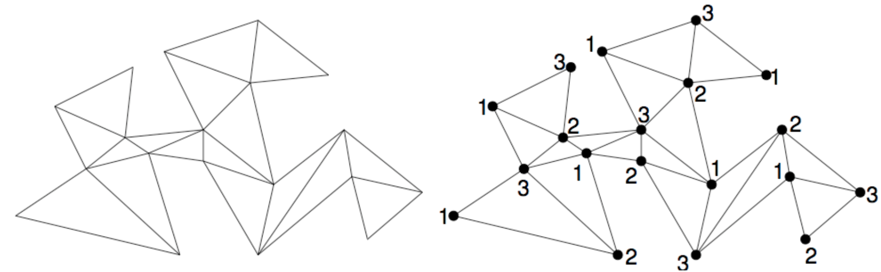
- Esiste un algoritmo (Chazelle, 1990) in $O(n)$ ma è talmente complicato da implementare che nella pratica non viene (ancora) impiegato.

- ***Problema della galleria d'arte***: Consideriamo galleria d'arte (su un piano), ci si chiede quante guardie servono in modo che ogni punto della galleria sia visibile da almeno una guardia.

- Formalizzazione: La pianta della galleria è un poligono semplice con n vertici; un punto P ***vede*** il punto Q se il segmento $\overline{PQ}$ è interamente contenuto nel poligono.

- Nota: il poligono non viene specificato, quindi la risposta deve valere per ogni poligono semplice con n vertici.

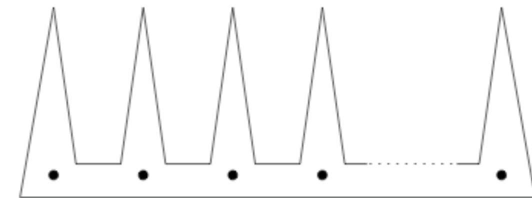
- ***Lemma***: Ogni triangolazione di un poligono è ***3-colorabile***, ovvero si possono colorare i vertici con tre colori diversi in modo che vertici adiacenti non abbiano lo stesso colore. (v. [3])



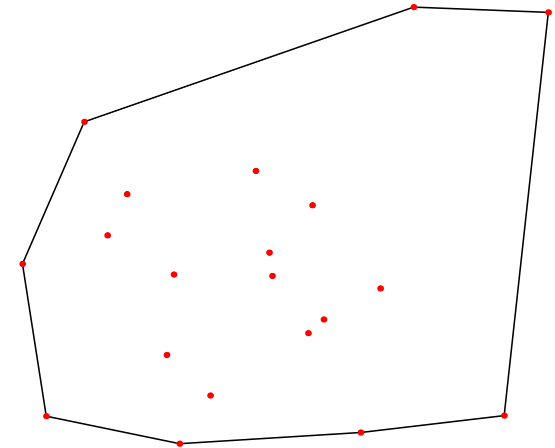
- ***Teorema*** (Art Gallery): Per sorvegliare un poligono semplice con n vertici sono sufficienti $\lfloor \frac{n}{3} \rfloor$ guardie.

- Dim. (Fisk) Il poligono si può triangolare, e la triangolazione è 3-colorabile. Uno dei colori deve essere usato non più di $\lfloor \frac{n}{3} \rfloor$ volte. Sia il rosso tale colore e si posizioni una guardia su ogni vertice rosso. Ciascun triangolo ha vertici di tutti in 3 colori, quindi ha sempre un vertice rosso, dunque ogni triangolo è sorvegliato. Poiché i triangoli coprono il poligono, tutto il poligono è sorvegliato.

- L'esempio del poligono a "pettine" in figura è un caso in cui $\lfloor \frac{n}{3} \rfloor$ guardie sono necessarie.



Guscio convesso



- Definizione: Il ***guscio convesso*** (*convex hull*) di un insieme finito di punti $\mathcal{S}$ in E^2, denotato da $conv(\mathcal{S})$, è il più piccolo insieme convesso di E^2 che contiene tutti i punti dati.

- ***Intuizione***: se i punti di $\mathcal{S}$ sono dei pioli su una tavola, li circondo con un elastico e lo lascio andare. Il poligono descritto dall'elastico teso attorno ad alcuni di essi è il contorno del guscio convesso.

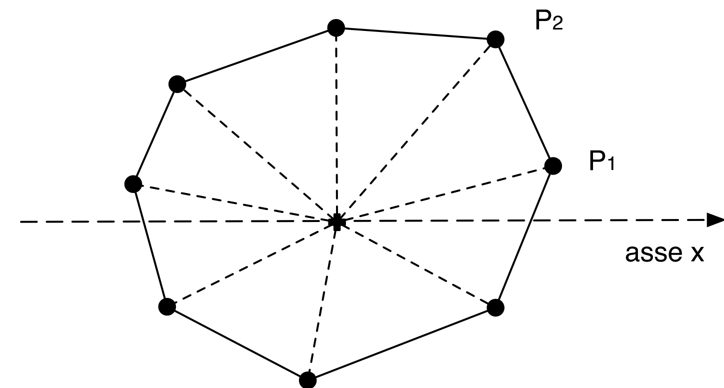
- La ***frontiera*** di $conv(\mathcal{S})$ – denotata da $\mathrm{CH}(\mathcal{S})$:

  i) è un poligono convesso

  ii) ha per vertici un sottoinsieme di punti $\mathcal{S}$, chiamati ***punti estremi***.

- Il punto i) segue dal seguente
- ***Teorema*** (Mc Mullen, Shephard): il guscio convesso $conv(\mathcal{S})$ di un insieme finito di punti $\mathcal{S}$ in $\mathbb{E}^2$ è un ***insieme poligonale (convesso) limitato***. Viceversa un insieme poligonale (convesso) limitato è il guscio convesso di un insieme finito di punti.
- ***Definizione***: Un punto P di un insieme convesso $\mathcal{S}$ è un punto ***estremo*** se non esistono due punti $A, B \in \mathcal{S}$ tali che P giaccia sul segmento aperto $\overline{AB}$.
- Il punto ii) deriva dalle seguenti osservazioni ([1] pag. 104).
  - L'insieme dei punti estremi $\mathcal{E}$ è il più piccolo sottoinsieme di $\mathcal{S}$ tale che $conv(\mathcal{E}) = conv(\mathcal{S})$
  - L'insieme dei punti estremi $\mathcal{E}$ coincide con i vertici di $CH(\mathcal{S})$.
- ***Problema:*** (Convex Hull) dato un insieme $\mathcal{S}$ di n punti nel piano, descrivere la frontiera del suo guscio convesso, $CH(\mathcal{S})$.
- Descrivere un poligono significa che bisogna fornire i suoi vertici in senso antiorario.
- Se ci si accontenta dell'insieme (non ordinato) dei vertici di $CH(\mathcal{S})$, si ha il problema Extreme Points, che richiede appunto l'elencazione dei punti estremi.

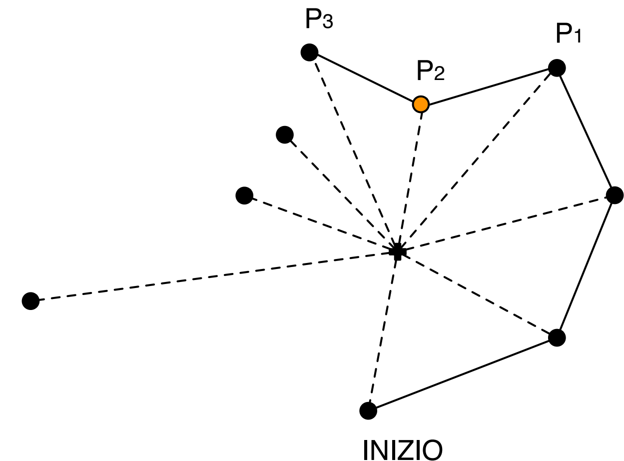
- Come ordinare i vertici di $\mathrm{CH}(\mathcal{S})$?
- ***Lemma 1***. Una semiretta uscente da un punto interno di un poligono convesso interseca il poligono esattamente in un punto (deriva dal teorema di Jordan, ([1] pag. 105)
- ***Lemma 2***. I vertici consecutivi di un poligono convesso sono ***ordinati per coordinata polare*** rispetto ad un qualunque punto interno. ([1] pag. 105)



- Come trovare i punti estremi?
- ***Teorema*** (Hadwiger-Debrunner) Un punto $P \in \mathcal{S}$ è ***estremo*** se e solo se non è contenuto in nessun triangolo che ha per vertici punti di $\mathcal{S} \setminus \{P\}$. ([1] pag. 104)
- Il teorema ci fornisce un ***test per eliminare punti che non sono estremi***. In principio ci sono $\binom{n}{3}$ (ovvero $O(n^3)$) triangoli da provare, per ciascun punto considerato, quindi un costo totale di $O(n^4)$. Non un grande passo avanti! Tuttavia viene fuori che non serve provarli tutti ...

Graham's Scan

- *Risolve* CONVEX HULL.
- ...infatti, dato un punto interno O, ogni punto non estremo deve essere interno ad un triangolo OPQ, dove P e Q sono vertici consecutivi del guscio convesso.
- L'*idea* di Graham è quella di ordinare i punti di $\mathcal{S}$ per coordinata polare rispetto ad un qualunque punto interno O e tutto si riduce ad ***una sola scansione dei punti in ordine polare***, durante la quale i punti interni vengono eliminati.
- Per trovare un punto interno O che funga da ***polo***, si prendano tre punti di $\mathcal{S}$ non allineati e si calcoli il baricentro del triangolo da essi formato.
- Siano P_1, P_2, P_3 tre punti consecutivi in senso antiorario. Se essi definiscono una ***svolta a destra***, allora P_2 ***non è un punto estremo*** poiché è interno al triangolo OP_1P_3 (per costruzione, giace dalla stessa parte di O rispetto a $\overline{P_1P_3}$ ed è compreso tra $\overline{OP_1}$ e $\overline{OP_3}$.)



- Questa osservazione è equivalente a dire che: percorrendo i vertici di un poligono convesso in senso antiorario, si fanno solo svolte a sinistra, ovvero che ***tutti i vertici di un poligono convesso sono convessi***.

- Come ***punto iniziale*** si prenda il punto di ordinata minima (il più in basso), e, a parità di ordinata, quello più a destra.

- Algoritmo di Graham (***Graham's scan***):

  1. Identifica un punto interno O.
  2. Ordina i punti di $\mathcal{S}$ per coordinate polari crescenti rispetto a O. Se più punti hanno la stessa coordinata angolare, elimina quelli più vicini a O.
  3. Inizia dal punto più in basso a destra.
  4. Scandisci i punti di $\mathcal{S}$ ordinati in senso antiorario esaminando terne di punti nel seguente modo:
     - Se P_1, P_2, P_3 definiscono una svolta a destra elimina P_2 e passa a controllare la terna P_0, P_1, P_3 (*backtrack*).
     - Se P_1, P_2, P_3 definiscono una svolta a sinistra avanza nella scansione, controllando la terna P_2, P_3, P_4.

- Viene comodo impiegare una ***pila*** (*stack*) per mantenere la descrizione di $CH(\mathcal{S})$. In termini più precisi, l'iterazione diventa:

```
Push(L,P1); Push(L,P2); Push(L,P3);
for i= 3 to n
   do while Orient(Next_to_top(L), Top(L), Pi) > 0
          do Pop(L);
      Push(L,Pi);
   end
```

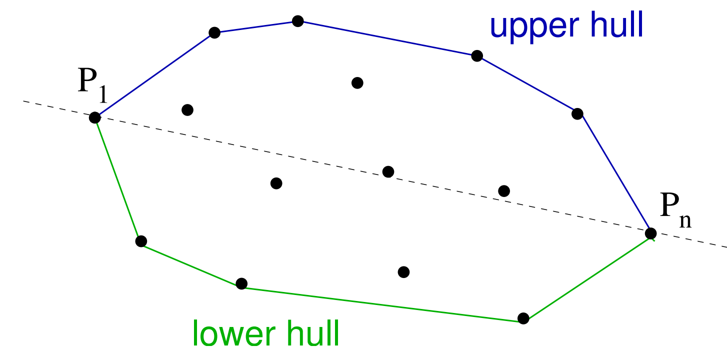
- ***Complessità***: dimostriamo che è $O(n \log n)$
  - Per ordinare gli n punti la complessità è $O(n \log n)$
  - Per ogni punto aggiunto durante la procedura la complessità è $O(k_i + 1)$, dove k_i sono i numeri di punti che devo eliminare durante quel passo
  - Sommando per tutti i punti si ha $\sum_i (k_i + 1) = n + \sum_i k_i$
  - Siccome ciascun punto può essere al più cancellato una volta sola, si ha $\sum_i k_i \leq n$ e dunque la complessità computazionale della fase di aggiunta dei punti è $O(n)$
  - Il termine dominante è l'ordinamento, dunque la complessità computazionale totale del metodo è $O(n \log n)$

- ***Variante 1*** (da [3]): invece del punto interno prendo come polo O il punto di partenza. Funziona ugualmente perché un poligono convesso si può triangolare "a ventaglio" tracciando le diagonali da un suo vertice.

- ***Variante 2*** (Andrews, da [1]): scandisco i punti per ascissa crescente.

- I punti vengono ordinati per ascissa crescente (più precisamente per ordine lessicografico sulle coordinate (x, y)).

- Il primo e l'ultimo punto sono sicuramente punti estremi. La retta che li contiene partiziona i punti di $\mathcal{S}$ in due sottoinsiemi.



- Si processano – come nel Graham's scan – separatamente i punti che stanno sopra e quelli che stanno sotto la retta, ottenendo rispettivamente il ***guscio superiore*** ed il***guscio inferiore***. Sono due catene poligonali monotone rispetto all'asse x, la cui unione è $CH(\mathcal{S})$.

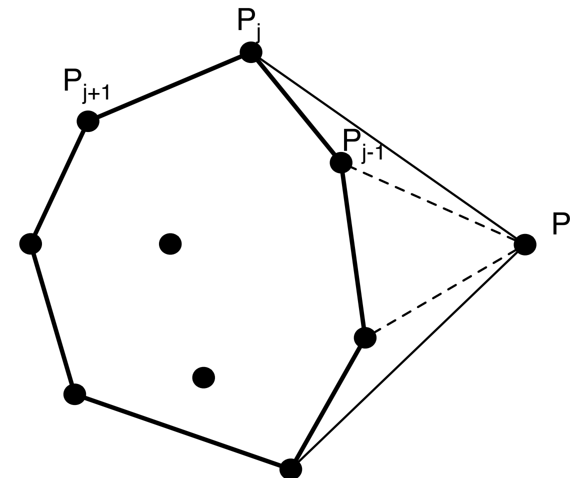
- È una specializzazione del metodo originale di Graham al caso in cui si prenda come polo O il punto a $-\infty$ sull'asse delle y.

- ***Vantaggi:***
  - Semplice da implementare
  - La complessità è ottimale (Intuitivamente: non si può fare meglio di $O(n \log n)$ perché i punti che descrivono il poligono devono essere ordinati).
- ***Svantaggi:***
  - Non si può generalizzare a n-d, perché il Lemma 2 vale solo nel piano.
  - Difficile da implementare in modo parallelo.

Algoritmo incrementale

- ***Risolve*** CONVEX HULL.
- L'idea (ricorrente in geometria computazionale) è quella di aggiungere un punto alla volta. L'algoritmo mantiene il guscio convesso dei punti già inseriti, quindi ad ogni inserimento si deve aggiornare il guscio convesso.

1. Si ordinano i punti di $\mathcal{S}$ secondo l'ordine lessicografico su (x, y). Inserendo i punti in quest'ordine si è garantiti che il punto da inserire non appartiene al guscio convesso dei precedenti.
2. Si costruisce il triangolo formato dai primi tre punti, che coincide anche con il loro guscio convesso.
3. Per aggiungere un punto P lo si collega al guscio convesso corrente con due nuovi spigoli, e si rimuovono quelli compresi tra i due. Tali spigoli prendono il nome di ***tangenti*** superiore ed inferiore.



- I punti di tangenza superiori/inferiori in ordine di aggiunta descrivono il guscio superiore/inferiore.

- Le due tangenti devono devono soddisfare due requisiti:
  (i) non intersecare il guscio convesso esistente
  (ii) definire svolte a sinistra con gli spigoli adiacenti (per mantenere la proprietà del guscio convesso).
- Siano $P_{j-1}, P_j, Pj+1$ tre punti consecutivi (in senso antiorario) sul guscio convesso corrente. Il punto P_j è di tangenza superiore se $\mathrm{orient}(P, P_{j-1}, P_j) < 0 \wedge \mathrm{orient}(P, P_j, P_{j+1}) > 0$ (analogamente per l'inferiore).
- Il controllo della condizione di tangenza per tutti i vertici nel guscio convesso corrente sortisce un algoritmo $O(n^2)$.
- Def. Un punto P_j è ***visibile*** da P_i, se il segmento $\overline{P_iP_j}$ non interseca altri spigoli.
- L'idea che consente di ridurre la complessità consiste nell'evitare di controllare anche i vertici del guscio convesso palesemente non visibili.
- A tal fine, si parte da un punto P_j sicuramente visibile (l'ultimo inserito lo è, grazie all'ordinamento) e si scorre il guscio convesso corrente nelle due direzioni (orario ed antiorario) a partire da P_j controllando la condizione di tangenza.

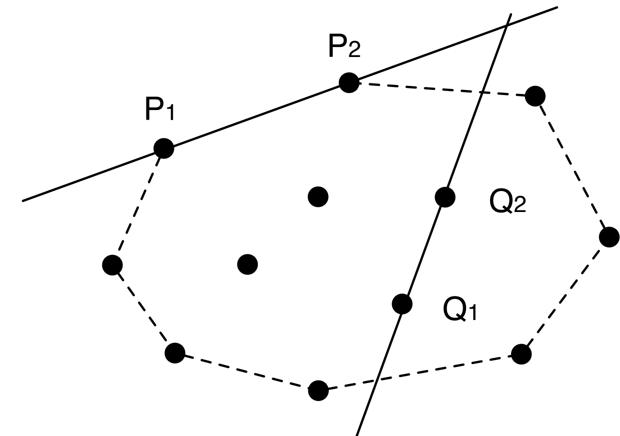
- ***Complessità***: gli n vertici vengono inseriti uno alla volta. Ad ogni inserimento vengono esaminati un certo numero di spigoli per il test di tangenza, due spigoli vengono inserirti ed altri vengono rimossi. L'osservazione chiave è che tutti gli spigoli coinvolti, considerati globalmente, non si intersecano, ovvero formano un grafo piano con n vertici e quindi sono $O(n)$. Domina quindi l'ordinamento dei punti, ed il costo totale è $O(n \log n)$.

- ***Vantaggi e svantaggi***: La complessità è ottimale, ed è semplice da implementare. Con una piccola modifica (ed allo stesso costo) calcola anche una triangolazione (vedremo).

- L'algoritmo si basa sulla costruzione ***incrementale***, che ritroveremo in altri contesti. I suoi elementi essenziali sono:
  - gli elementi che costituiscono i dati (punti) vengono aggiunti uno alla volta
  - il costruendo oggetto geometrico (il guscio convesso) viene aggiornato dopo ogni inserimento.
  - I dati possono venire inseriti in un particolare ordine, oppure a caso. In quest'ultimo caso si ha un approccio incrementale randomizzato.

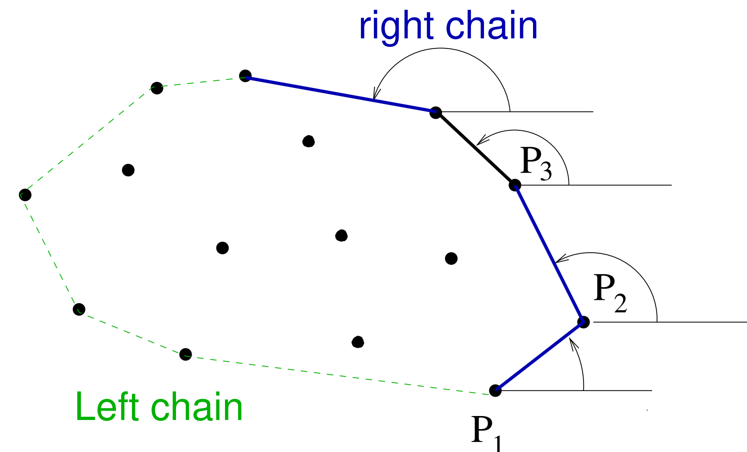
Jarvis's march (gift wrapping)

- ***Risolve*** CONVEX HULL.
- Il Graham's scan si concentra sul determinare i vertici di $\mathrm{CH}(\mathcal{S})$.
- Il ***Jarvis's march*** – che vediamo adesso – invece punta ad individuare gli spigoli di $\mathrm{CH}(\mathcal{S})$, grazie al seguente:
- ***Teorema*** (Stoer-Witzgall)
  Un segmento che ha per estremi due punti di $\mathcal{S}$ è uno ***spigolo*** di $\mathrm{CH}(\mathcal{S})$ se e solo se tutti gli altri punti di $\mathcal{S}$ ***giacciono dalla stessa parte*** della retta contenente il segmento (retta compresa).



- Ovvero: la retta che passa per due vertici consecutivi di $\mathrm{CH}(\mathcal{S})$ lascia tutti gli altri punti di $\mathcal{S}$ dalla stessa parte.
- Nella figura, $\overline{P_1P_2}$ è uno spigolo di $\mathrm{CH}(\mathcal{S})$ perché tutti i punti di $\mathcal{S}$ giacciono dalla stessa parte, mentre $\overline{Q_1Q_2}$ non lo è perché ci sono punti da entrambe le parti.

- L'algoritmo che viene subito in mente è il seguente:
  1. Per ogni coppia di punti P_i P_j dell'insieme si calcola $\text{orient}(P_i, P_j, P_k)$ $\forall k \neq i, j$
  2. Se tale valore è positivo $\forall k \neq i, j$ allora i due punti sono estremi.
- ***Complessità***: si considerano $\binom{n}{2}$ 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.
- L'algoritmo non è ottimale, ma si può migliorare...
- ***Osservazione 1.*** Una volta stabilito che il segmento $\overline{P_1 P_2}$ è uno spigolo $CH(\mathcal{S})$, il prossimo spigolo deve avere P_2 come estremo.
- ***Osservazione 2.***
  L'altro estremo P_3 è il punto che ha il minor angolo polare rispetto a P_2 (intuizione: si traccia la retta orizzontale che passa per P_2 tale punto e la si ruota in senso antiorario fino a quando non incontra un punto).



- Allora l'algoritmo di Jarvis (***Jarvis's march***) è il seguente:
  1. Si parte dal punto P_1 più in basso, a destra, (come in Graham's scan) e quindi è sicuramente un punto estremo.
  2. Il vertice consecutivo è il punto di $\mathcal{S}$ che ha la minima coordinata angolare in un sistema polare di coordinate centrato in P_1 (ovvero forma con l'asse delle ascisse l'angolo minore).
  3. Si ripete il passo 2 fino a raggiungere il punto più in alto, a sinistra.
- Si ottiene così una catena poligonale monotona rispetto all'asse y che rappresenta la parte destra di $\mathrm{CH}(\mathcal{S})$.
- Per ottenere la catena di sinistra si procede in modo simmetrico, con la ***direzione dell'asse*** x ***invertita***.
- L'unione della catena di destra e di quella di sinistra è $\mathrm{CH}(\mathcal{S})$.
- Ricordiamo che non occorre calcolare l'angolo polare (con operazioni trigonometriche) ma si usa la funzione orient().
- L'algoritmo si può ben visualizzare come l'incartamento di un pacco (*gift-wrapping*) per ovvi motivi.

- ***Complessità***: il costo della determinazione del punto successivo è $O(n)$, e questo viene fatto per ogni vertice del guscio convesso. Se tali vertici sono h, allora la complessità computazionale dell'algoritmo è $O(nh)$. Nel caso pessimo si arriva a $O(n^2)$ poiché si ha $h = O(n)$

- ***Vantaggi e svantaggi***: è un algoritmo ***output sensitive***, ovvero, la cui complessità dipenda anche dalla dimensione dell'output. Se solo una piccola frazione di punti sono estremi, è molto veloce. Tuttavia la complessità nel caso pessimo non è ottimale.

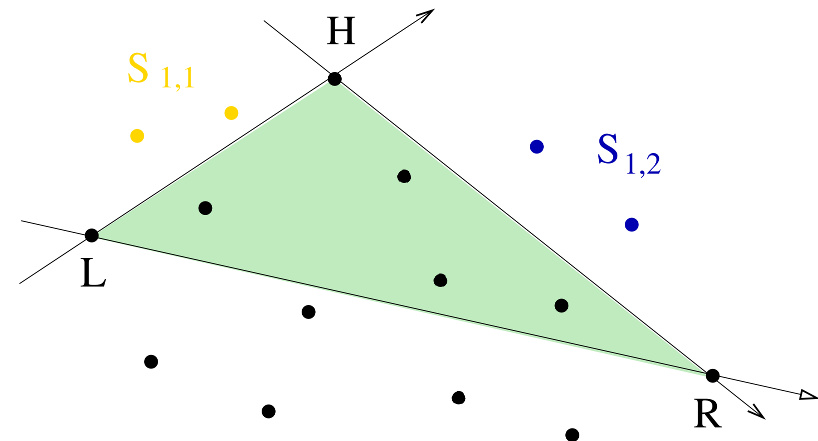
- Quanti punti estremi possiamo aspettarci?

- ***Teorema***. Dati n punti uniformemente distribuiti in un quadrato unitario, il valore atteso del numero di punti estremi è $O(\log n)$. (dim. su [6]).

- Esiste un algoritmo *output sensitive* che risolve CONVEX HULL in $O(n \log h)$.

QuickHull

- ***Risolve*** EXTREME POINTS.
- Per ottenere CONVEX HULL devo ordinare polarmente i punti.
- L'idea è simile al QuickSort, ovvero si basa sull'approccio "divide-et-impera".
- Si considerano i punti di ascissa minima e massima, L (*leftmost*) ed R (*rightmost*). Essi appartengono a $\mathrm{CH}(\mathcal{S})$.
- La retta che li congiunge partiziona l'insieme $\mathcal{S}$ in due sottoinsiemi $\mathcal{S}_1$ e $\mathcal{S}_2$,.
- Sia H il punto in $\mathcal{S}_1$ di massima distanza dalla retta passante per L ed R. Se ce ne sono più d'uno, si prenda quello che massimizza l'angolo $L\hat{H}R$, ovvero quello più a sinistra.



- Si vede facilmente che H è un punto estremo. Infatti, per costruzione non ci possono essere punti al di sopra della retta parallela ad LR passante per H. Ci possono essere altri punti sulla retta stessa, ma avendo scelto H come il punto più a sinistra, esso non può essere combinazione convessa degli altri.

- Si considerino ora le due rette orientate da L ad H e da H a R rispettivamente. Si osservi che:
  - non esistono punti ***a sinistra di entrambe*** (perché H è estremo)
  - quelli che giacciono ***a destra di entrambe*** sono interni al triangolo LRH e si possono eliminare dalla considerazione (perché sono interni al guscio convesso)
  - quelli che giacciono ***a destra di una retta ed a sinistra dell'altra*** costituiscono rispettivamente i due insiemi $\mathcal{S}_{1,1}$ ed $\mathcal{S}_{1,2}$ sui quali la procedura si attiva ricorsivamente.
- La prima partizione si ottiene prendendo come retta partizionante la retta verticale passante per L. In questo modo il punto più lontano è proprio R e la ricorsione può iniziare.
- Come in QuickSort – ed al contrario di MergeSort – il lavoro si svolge nella fase "dividi", mentre la fase "combina" è banale.

- Pseudocodice di ***QuickHull*** (da [5])

```
Algorithm QUICKHULL (S,L,R)
begin
  if (S={L,R}) then return ([L,R]);
  /* condizione di terminazione del processo ricorsivo */
  else  /* S contiene piu' di due punti */
  begin
    H := FURTHEST(S,L,R);
    /* H e' il punto piu' distante dalla retta LR secondo quanto spiegato prima */
    S1 := punti di S giacenti a sinistra della retta LH;
    S2 := punti di S giacenti a sinistra della retta HR;
    return ( CONCATENATE (QUICKHULL(S1,L,H), QUICKHULL(S2,H,R) );
    /* CONCATENATE concatena le due liste eliminando il punto H dalla seconda lista *
   else
end /* QUICKHULL */

/* Chiamata /*
L = (X0,Y0) /* punto di minima ascissa */
R = (X0,Y0-EPS) /* punto fittizio per ottenere la retta verticale per L */
Hull = QUICKHULL(S,L,R);
```

- ***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 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:

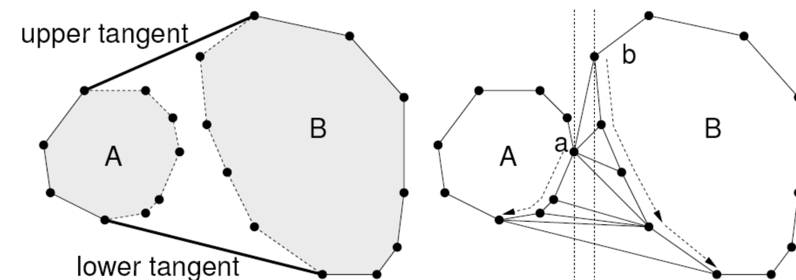
$$T(n) = \begin{cases} 1 & \text{se } n = 2 \\ T(n_1) + T(n_2) + n & \text{altrimenti} \end{cases}$$

    dove n_1 ed n_2 sono e cardinalità di $\mathcal{S}_1$ ed $\mathcal{S}_2$ rispettivamente.
  - Se assumiamo $\max(n1, n2) < \alpha n$ 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 (perché spezza il problema in sottoproblemi). La complessità nel caso peggiore non è ottimale, ma e nella pratica è veloce.

MergeHull

(cenni)

- QuickHull è un ottimo esempio del paradigma divide-et-impera, ma resta poco soddisfacente il fatto di non poter controllare la dimensione dei due sottoproblemi, e quindi di garantire una complessità ottimale anche nel caso peggiore.
- L'idea del MergeHull (Shamos) è quella di partizionare $\mathcal{S}$ con una retta verticale in due sottoinsiemi $\mathcal{S}_{\mathrm{L}}$ ed $\mathcal{S}_{\mathrm{R}}$ contenenti *lo stesso numero di punti*, calcolare $\mathrm{CH}(\mathcal{S}_{\mathrm{R}})$ $e\mathrm{CH}(\mathcal{S}_{\mathrm{L}})$ ricorsivamente e quindi fondere (unire) i due risultati per ottenere $\mathrm{CH}(\mathcal{S})$.
- Come nel MergeSort, la partizione è perfettamente bilanciata ed è facile da ottenere, mentre la difficoltà è spostata nella fase "combina".
- L'idea è la seguente: si parte collegando il punto più a destra di $\mathcal{S}_{\mathrm{L}}$ con quello più a sinistra di $\mathcal{S}_{\mathrm{R}}$ e ci si muove verso l'alto e verso il basso alla ricerca della tangente superiore e di quella inferiore, in modo analogo all'algoritmo incrementale ([1], pg. 117).



- ***Complessità***. L'unione dei due gusci convessi costa $O(n)$ e MergeHull complessivamente richiede $O(n \log n)$ (la ricorrenza è la stessa del MergeSort).

- ***Vantaggi e svantaggi***: la complessità è ottimale, è parallelizzabile (come QuickHull), ma è più complesso di QuickHull da implementare (a causa della fase di unione).

Guscio convesso in 3D

(cenni)

- La generalizzazione dell'algoritmo di Jarvis (*gift-wrapping*) è immediata. La complessità è $O(nf)$ dove f sono le facce del guscio. Nel caso peggiore è $O(n^2)$. È tra i migliori in alte dimensioni.
- Anche ***QuickHull*** si generalizza al 3D, mantenendo la complessità di $O(n^2)$ nel caso peggiore. Nella pratica tuttavia si comporta molto bene, tanto che il software Qhull è tra i più usati (p. es. dentro MATLAB, Octave e Blender).
- L'algoritmo incrementale si generalizza, raggiungendo $O(n^2)$ nella caso peggiore ed un costo atteso $O(n \log n)$ nella versione randomizzata.
- Anche l'algoritmo MergeHull si generalizza al 3D, e raggiunge la complessità ottimale di $O(n \log n)$ (ma è difficile da implementare, per questo poco usato).

Applicazioni del guscio convesso

- ***Collisioni*** (grafica). Un problema tipico nelle applicazioni interattive di grafica è il calcolo delle collisioni tra oggetti. Se non si richiede una precisione elevata (caso frequente) si può usare il guscio convesso degli oggetti, invece che la loro descrizione precisa, per calcolare le collisioni, semplificando molto il problema.
- Il guscio convesso serve anche per il calcolo del più piccolo rettangolo che contiene un insieme di punti.
- ***Collisioni*** (robotica). Nella pianificazione del moto di un robot, se il guscio convesso del robot non urta gli ostacoli non lo fa neanche il robot.
- Inoltre il guscio convesso è un oggetto che gioca un ruolo centrale nella geometria computazionale, essendo legato da relazioni di dualità ad altri importanti problemi (intersezione di semispazi e triangolazione di Delaunay).

Intersezioni

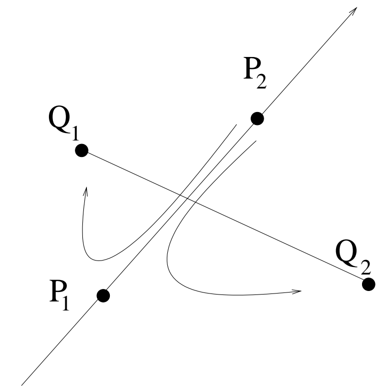
- Motivazione: due oggetti non possono occupare lo stesso posto allo stesso tempo.
- Applicazioni:
  - rimozione delle linee e delle superfici nascoste in Grafica (un oggetto ne oscura un altro se le loro proiezioni si intersecano).
  - Separabilità lineare di due insiemi di punti in Pattern Recognition
  - 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'algoritmo di forza bruta richiede $O(n^2)$ tempo, poiché considera ciascuna coppia di segmenti. Se tutti i segmenti si intersecano è ottimo, perché solo l'elencazione dell'output richiede $O(n^2)$, essendo $\binom{n}{2}$ le intersezioni nel caso peggiore.
- Dunque considerando la complessità nel caso pessimo non si può fare meglio di così. Nella pratica, però, ciascun segmento ne interseca pochi altri.
- Cerchiamo quindi un algoritmo ***output sensitive***, ovvero il cui tempo di esecuzione dipenda anche dalla dimensione dell'output.
- Un'analisi di complessità più raffinata, che tenga in considerazione sia la dimensione dell'input che quella dell'output è necessaria.
- Prima vediamo come stabilire se *due* segmenti di intersecano.

Test di intersezione tra due segmenti

- ***Problema***: determinare se due segmenti in E^2 si intersecano.
- Un segmento interseca una retta sse i suoi estremi giacciono da parti opposte rispetto alla retta.
- Per esempio, il segmento $\overline{Q_1Q_2}$ interseca la retta passante per P_1 e P_2 se Q_1 e Q_2 giacciono da parti opposte della retta, ovvero se $\text{orient}(P_1, P_2, Q_1)$ e $\text{orient}(P_1, P_2, Q_2)$ 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 $\overline{P_1P_2}$ e $\overline{Q_1Q_2}$ si intersecano se il segmento $\overline{Q_1Q_2}$ interseca la retta passante per P_1 e P_2 e contemporaneamente il segmento $\overline{P_1P_2}$ interseca la retta passante per Q_1 e Q_2 ovvero se

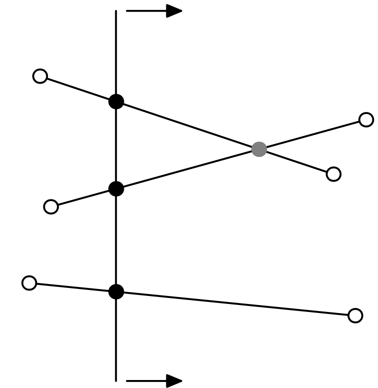
$$\text{orient}(P_1, P_2, Q_1) \cdot \text{orient}(P_1, P_2, Q_2) < 0 \quad \text{e} \quad \text{orient}(Q_1, Q_2, P_1) \cdot \text{orient}(Q_1, Q_2, P_2) < 0$$

- ***Complessità***: costante ($O(1)$) poiché il numero di operazioni è fissato.
- Casi particolari vanno gestiti a parte (v. [5]).

Algoritmo plane sweep

(Bentley, Ottmann, 1979)

- ***Risolve*** LINE-SEGMENT INTERSECTION.
- 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.
- Per non complicare l'esposizione supporremo vere le seguenti condizioni semplificatrici:
  - Non ci sono segmenti verticali
  - I segmenti si possono intersecare in un solo punto
  - 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 per i quali esiste una linea retta verticale che interseca entrambe.



- Per trovare
  tali coppie simuliamo il passaggio di una linea verticale (***sweep line***) da sinistra a destra che "spazzola" l'insieme dei segmenti.

- Questo però non basta per ottenere un algoritmo output sensitive, perché esistono situazioni dove si effettua un numero quadratico di test a fronte di un piccolo numero di intersezioni (p.es. segmenti orizzontali tutti intersecati da una retta verticale.)

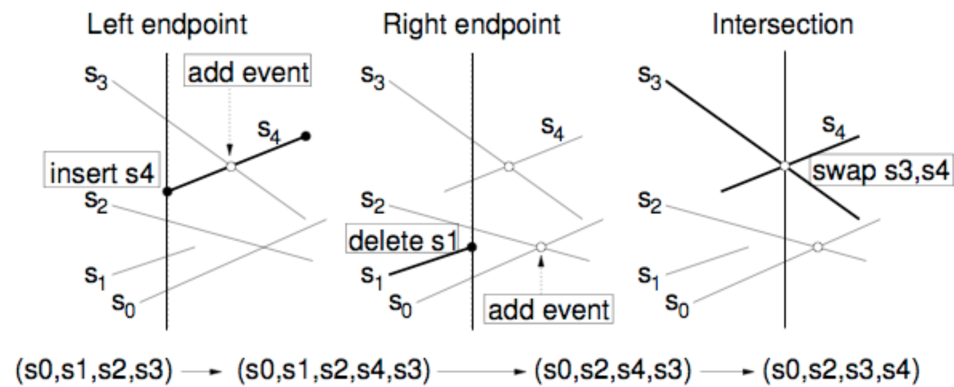
- ***Idea 2***: Bisogna includere anche la nozione di vicinanza nella direzione verticale: i segmenti che intersecano la *sweep line* vengono ordinati dal basso verso l'alto, e ***verranno controllati solo segmenti adiacenti***. La correttezza di questo approccio si basa sul seguente

- ***Lemma***: se due segmenti s_i ed s_j si intersecano in un punto (x, y) allora esiste un'ascissa $x^* < x$ per cui s_i ed s_j sono adiacenti lungo la *sweep line* (e dunque vengono controllati) (dim. su [6]).

- Lo ***stato*** della *sweep line* è la sequenza dei segmenti che la intersecano, ordinata secondo la coordinata y della intersezione.
- Lo stato della *sweep line* cambia solo in corrispondenza di un ***evento***, ovvero
  - quando la *sweep line* raggiunge ***un estremo di un segmento***
  - quando incontra ***un punto di intersezione tra due segmenti***
- Si noti che mentre gli estremi dei segmenti sono noti a priori, le loro intersezioni non lo sono (naturalmente!). Verranno scoperte man mano che la "spazzolata" procede.
- L'algoritmo funziona perché i punti di intersezione vengono scoperti come eventi *prima* che la *sweep line* li sorpassi (per il Lemma).
- Ovvero: ***non ci sono intersezioni non rilevate alla sinistra della linea***.
- Dunque, ricapitolando:
  - La *sweep line* non si muove con continuità ma su un insieme di posizioni discrete, corrispondenti agli eventi.
  - In queste posizioni viene aggiornato lo stato della *sweep line* e si calcolano le intersezioni
  - Il calcolo delle intersezioni coinvolge solo i segmenti adiacenti nello stato della *sweep line*.

- L'algoritmo si appoggia su due strutture dati.
- ***La coda degli eventi*** $\mathcal{E}$: contiene gli eventi noti a destra della *sweep line* ordinati per x crescente. Ogni elemento della coda deve specificare che tipo di evento sia e quali segmenti coinvolga. Tale struttura dati deve permettere l'inserzione di nuovi eventi (non ancora presenti nella coda) e l'estrazione dell'evento iniziale della coda. Si può usare un albero binario bilanciato che consente di effettuare le operazioni necessarie in $O(\log |\mathcal{E}|)$.
- ***Stato della* sweep line** $\mathcal{L}$: contiene i segmenti intersecati dalla *sweep line*, ordinati per y crescente (che cambia con il muoversi della linea). Si deve poter eliminare un segmento da $\mathcal{S}$, inserirne uno nuovo, scambiare l'ordine di due segmenti consecutivi e determinare il precedente ed il successivo di ciascun elemento. Si può usare, di nuovo, un albero binario di ricerca bilanciato, che consente di effettuare le operazioni necessarie in $O(\log |\mathcal{L}|)$.

Vediamo quindi *l'algoritmo di plane sweep* completo:



1. Siano $\{s_1, \ldots, s_n\}$ gli n segmenti da analizzare, rappresentati mediante gli estremi.
2. Si parte con la *sweep line* tutta a sinistra, il suo stato $\mathcal{L}$ è vuoto. Si inseriscono nella coda degli eventi $\mathcal{E}$ tutti gli estremi dei segmenti $\{s_1, \ldots, s_n\}$;
3. Fintanto che $\mathcal{E} \neq \emptyset$ si estrae l'evento successivo. Ci sono tre casi:
   i) ***Estremo sinistro di un segmento:*** si aggiunge il segmento a $\mathcal{L}$ e si fa un test di intersezione con il suo predecessore e con il suo successore.
   ii) ***Estremo destro di un segmento:*** si elimina il segmento da $\mathcal{L}$ e si fa un test di intersezione tra il suo (ex) predecessore ed il suo (ex) successore.
   iii) ***Punto di intersezione:*** si scambiano in $\mathcal{L}$ i due segmenti coinvolti nell'intersezione e per quello che finisce sopra/sotto si fa un test di intersezione con il suo predecessore/successore.
4. Si inseriscono in $\mathcal{E}$ gli eventuali punti di intersezione. Quelli nuovi (che non sono già presenti) vengono "stampati" in uscita.

- ***Complessità***: Il lavoro fatto dall'algoritmo è dominato dal tempo speso nell'aggiornamento delle strutture dati (per il resto sarebbe un tempo costante per ogni evento). Gli eventi sono al più $2n + \ell$, dove ℓ è il numero di intersezioni ($\ell \leq \binom{n}{2}$). Quindi la *sweep line* percorre al più $2n + \ell = O(n + \ell)$ stati. Le operazioni su $\mathcal{L}$ costano al più $O(\log n)$ (perché $\mathcal{L}$ contiene al più n elementi). L'inizializazzione di $\mathcal{E}$ comporta l'ordinamento di $2n$ elementi, quindi costa $O(n \log n)$. Le operazioni su $\mathcal{E}$ possono costare al più $O(\log(2n + \ell)) = O(\log(n^2)) = O(\log n)$. In totale la complessità è $O((n + \ell) \log n)$.

- Questa complessità non è ottimale. Esiste un algoritmo ottimale di Chazelle e Edelsbrunner che richiede $O(\ell + n \log n)$.

- ***Problema:*** (LINE-SEGMENT INTERSECTION TEST) dati n segmenti di retta nel piano, stabilire 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 $O(n \log n)$.

- L'algoritmo procede come quello appena visto, ma la coda degli eventi $\mathcal{E}$ contiene solo i $2n$ estremi dei segmenti. L'algoritmo termina non appena una intersezione viene rilevata.

- ***Osservazione***: Questo algoritmo trova, se ne esiste una, l'intersezione più a sinistra. Infatti la coda degli eventi per la regione a sinistra della prima intersezione è identica alla coda degli eventi dell'algoritmo originale (ci sono solo estremi di segmenti), e quindi, fino a quel punto, funziona come l'algoritmo originale.

- ***Complessità***: L'analisi procede come per l'algoritmo originale, ma manca il termine ℓ poiché la coda degli eventi contiene esattamente $2n$ elementi. Quindi, in totale la complessità è $O(n \log n)$. Si può dimostrare che è ottimale.

Pseudocodice (da [5]).

```
Algorithm LINE_INTERSECTION_TEST
  ordina i 2N punti estremi dei segmenti;
  inserisci tali punti ordinati in array E[1..2N];
  for i=1 to 2N do
    p := E[i];  /* p e' l'evento corrente */
    s := segmento di cui p e' estremo;
    if p estremo sinistro
           INSERT(s,L);
           s1 := ABOVE(s,L);
           s2 := BELOW(s,L);
           if (s1 interseca s) then return "SI"
           if (s2 interseca s) then return "SI"
      else /* p estremo destro */
           s1 := ABOVE(s,L);
           s2 := BELOW(s,L);
           if (s1 interseca s2) then return "SI"
           DELETE(s,L);
      end /* if */
  end /* for */
```

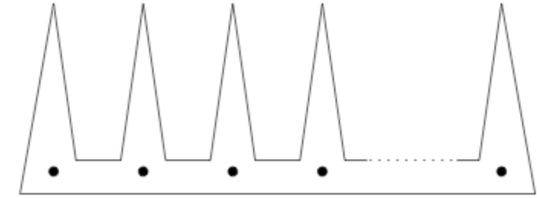
- La metodologia ***plane sweep*** introdotta da questo algoritmo ricorre più volte nell'ambito della geometria computazionale.
- I suoi elementi essenziali sono:
  - La linea (*sweep line*) che spazzola il piano fermandosi in certi punti speciali chiamati eventi;
  - L'invariante: il problema è risolto nel semipiano già spazzolato dalla *sweep line*;
  - L'intersezione della *sweep line* con i dati del problema contiene tutta l'informazione rilevante per la prosecuzione della spazzolata;
  - La coda degli eventi, che definisce le posizioni di fermata della *sweep line* (può essere aggiornata dinamicamente);
  - Lo stato della *sweep line* che contiene una descrizione adeguata della intersezione della linea con i dati.

- ***Problema:*** (SIMPLICITY TEST) dato un poligono con n spigoli, stabilire se è semplice.
- Si risolve controllando se gli spigoli si intersecano (eccetto che nei vertici). Risolvo quindi LINE-SEGMENT INTERSECTION TEST in $O(n \log n)$.

Intersezioni di poligoni

- ***Problema:*** (POLYGON INTERSECTION TEST) dati due poligoni semplici $\mathcal{P}$ con k vertici e $\mathcal{Q}$ con m vertici, decidere se si intersecano.
- Osservazione: $\mathcal{P}$ e $\mathcal{Q}$ si intersecano se e solo se vale una delle seguenti condizioni i) $\mathcal{P}$ è contenuto in $\mathcal{Q}$, ii) $\mathcal{Q}$ è contenuto in $\mathcal{P}$, iii) qualche spigolo di $\mathcal{P}$ interseca qualche spigolo di $\mathcal{Q}$.
- Perché $\mathcal{P}$ sia interno a $\mathcal{Q}$ è necessario che *tutti* i vertici di $\mathcal{P}$ siano interni a $\mathcal{Q}$, dunque per *escludere* questo caso basta risolvere una istanza del problema di POLYGON INCLUSION (che vedremo) usando un qualunque vertice di $\mathcal{P}$, con un costo $O(m)$. La stessa cosa si fa per escludere che $\mathcal{Q}$ sia interno a $\mathcal{P}$, con costo $O(k)$.
- Escluso che un poligono sia interno all'altro, passo a verificare la condizione iii), risolvendo un problema di LINE-SEGMENT INTERSECTION TEST, con un costo computazionale di $O(n \log n)$ con $n = k + m$.
- ***Complessità***: $O((k+m)\log(k+m))$
- Per risparmiare operazioni nei casi concreti, può essere bene effettuare un test preliminare di intersezione tra i rettangoli minimali (*bounding box*) che contengono i due poligoni.

- ***Problema:*** (POLYGON INTERSECTION) dati due poligoni semplici $\mathcal{P}$ con k vertici e $\mathcal{Q}$ con m vertici, determinare la loro intersezione.
- L'intersezione di due poligoni semplici può constare di uno o più poligoni.
- L'intersezione di due poligoni semplici può avere $O(km)$ vertici (es. due poligoni "a pettine").



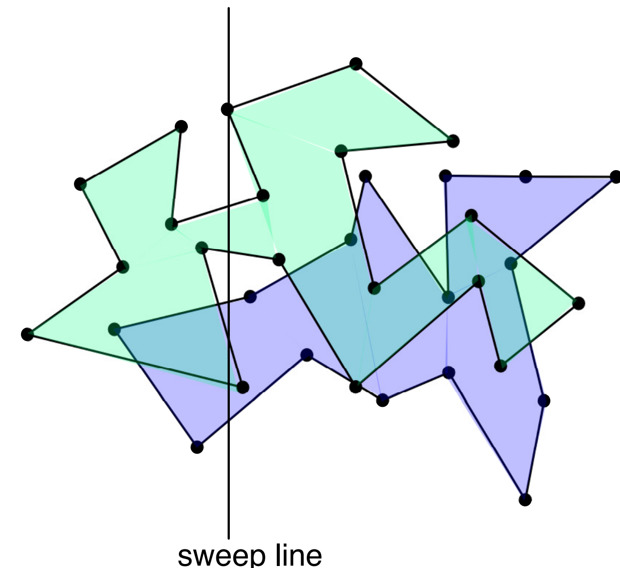
- Sebbene nel caso peggiore non si possa fare meglio di una complessità quadratica, un algoritmo *output sensitive* può dare buoni risultati in pratica.
- L'intersezione di due poligoni si riconduce a calcolare intersezione dei loro spigoli.

• Possiamo modificare l'algoritmo *plane sweep* di Bentley-Ottmann come segue.

Algoritmo plane sweep

(Cenni, da [3])

- ***Risolve*** POLYGON INTERSECTION.
- Modifica del *plane sweep* di Bentley-Ottmann.
- Manteniamo lungo la *sweep line* un indicatore di ***status*** per ciascuno dei segmenti nei quali è divisa dalle intersezioni con gli spigoli di $\mathcal{P}$ e $\mathcal{Q}$.
- Lo status può essere: esterno a $\mathcal{P}$ e $\mathcal{Q}$, interno a $\mathcal{P}$ ma esterno a $\mathcal{Q}$, interno a $\mathcal{Q}$ ma esterno a $\mathcal{P}$, interno a $\mathcal{P}$ e $\mathcal{Q}$.
- Lo status viene mantenuto aggiornato mentre la linea procede la spazzolata.
- Contemporaneamente vengono accresciute le catene poligonali che formeranno alla fine $\mathcal{P} \cap \mathcal{Q}$.



- ***Complessità***. La complessità rimane quella del *plane sweep* di Bentley-Ottmann, quindi l'intersezione di due poligoni semplici con n vertici in totale, si calcola in $O((n + \ell) \log n)$, dove ℓ è il numero di intersezioni tra gli spigoli dei due poligoni.

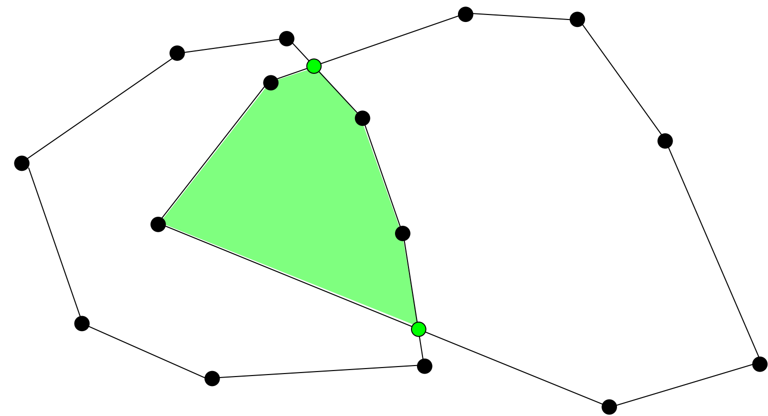
- ***Problema:*** (Convex Polygon Intersection) dati due poligoni convessi $\mathcal{P}$ con k vertici e $\mathcal{Q}$ con m vertici, determinare la loro intersezione.

- In questa parte possiamo confondere il "poligono" con la regione poligonale convessa formata dal poligono unito al suo interno.

- ***Teorema***. L'intersezione di due poligoni convessi $\mathcal{P}$ con con k vertici e $\mathcal{Q}$ con m vertici è un poligono convesso con al più $k + m$ vertici.

- Dim. L'intersezione delle due regioni poligonali delimitate da $\mathcal{P}$ e $\mathcal{Q}$ è l'intersezione dei $k + m$ semipiani definiti dalle rette orientate che contengono gli spigoli di $\mathcal{P}$ e $\mathcal{Q}$.

- Il bordo di $P \cap Q$ consiste di catene alternate di vertici dei due poligoni inframmezzate da punti in cui i bordi si intersecano.

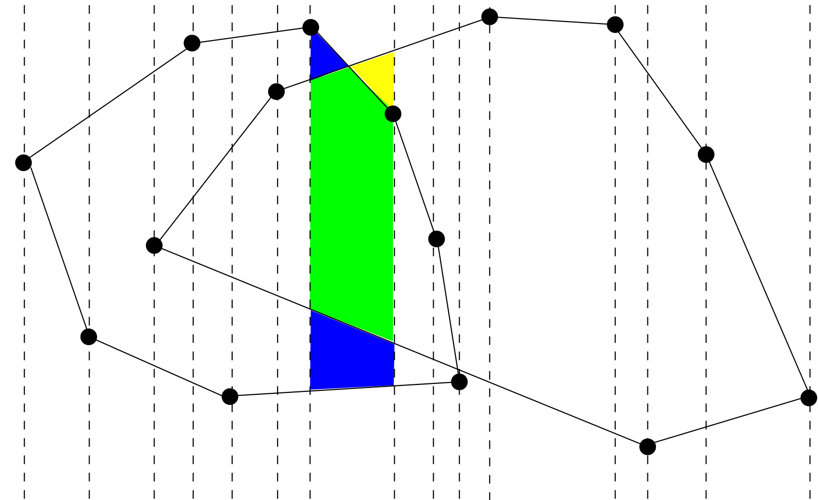


- Viene subito in mente il seguente algoritmo (costo $O(km)$): procedi lungo P controllando l'intersezione dello spigolo corrente con tutti quelli di Q. Si può fare in tempo lineare ...

Metodo delle Strisce

(Shamos e Hoey, 1976)

- ***Risolve*** CONVEX POLYGON INTERSECTION.
- Siano $\mathcal{P}$ e $\mathcal{Q}$ i due poligoni convessi di k ed m spigoli rispettivamente.



- Si suddivide il piano in strisce (***slab***) verticali passanti per i vertici di $\mathcal{P}$ e $\mathcal{Q}$. Ce ne sono $m + k - 1$.
- Ordino le strisce per ascissa crescente. Dato che i vertici di $\mathcal{P}$ e $\mathcal{Q}$ sono separatamente ordinati, per metterli assieme il costo è $O(k + m)$ (v. sotto).
- L'intersezione non vuota di una striscia con uno dei due poligoni è un trapezoide. Si calcola in tempo costante, se la struttura dati permette di accedere ai due spigoli incidenti su un dato vertice in $O(1)$.

- All'interno di ciascun striscia devo calcolare l'intersezione di due trapezoidi, che è un poligono con al più 6 spigoli. Costo $O(1)$.

- Alla fine unisco i poligoni intersezione provenienti da ciascuna striscia con una passata a costo $O(k + m)$ (devo solo eliminare il lato verticale in comune).

- ***Dettaglio***: come ottenere la lista ordinata dei vertici dei due poligoni in tempo lineare.

- Si spezza ciascuno dei due poligoni in due catene poligonali dal minimo al massimo lessicografico su (x, y). Le catene inferiori vengono invertite, ottenendo così quattro catene monotone rispetto alle ascisse ed ordinate per ascissa crescente. Costo lineare nel numero di vertici.

- Si fondono a due a due le catene come i vettori ordinati in MergeSort, costo lineare nella lunghezza delle catene.

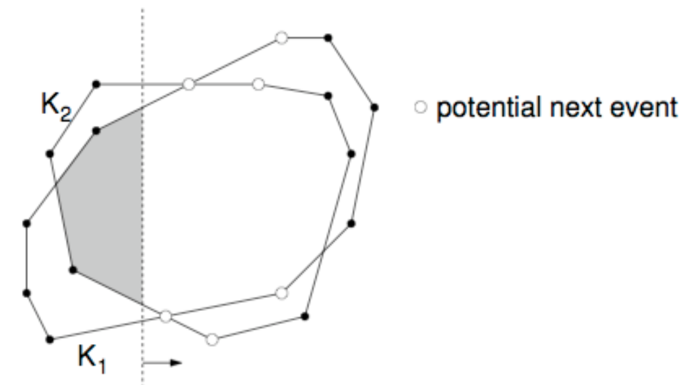
- ***Complessità***: sia l'ordinamento che la scansione delle strisce è lineare nel numero complessivo dei vertici, dunque $O(k + m)$.

- In questo algoritmo abbiamo visto applicata una ***tecnica*** comune in Geometria Computazionale, che consiste nella creazione di oggetti geometrici più semplici (i trapezi), risoluzione del problema semplificato per tali oggetti (intersezione di trapezi), fusione dei risultati parziali.

Algoritmo plane sweep

(Cenni, da [6])

- ***Risolve*** CONVEX POLYGON INTERSECTION.
- L'intersezione di due poligoni convessi (o anche di due regioni poligonali convesse, eventualmente illimitate) si può calcolare in tempo lineare semplificando l'algoritmo di *plane sweep* per l'intersezione di segmenti di Bentley-Ottmann.
- Poiché la *sweep line* interseca i due poligoni in al più 4 punti (per la convessità) le operazioni sullo stato della *sweep line* hanno costo costante (e non serve una struttura dati dinamica).



- Anche le operazioni per determinare il prossimo evento hanno costo costante: basta guardare gli estremi destri dei 4 segmenti correnti e le loro eventuali intersezioni (al più 4). Non serve la coda degli eventi.
- La *sweep line* percorre $O(k + m)$ stati ed ogni volta effettua operazioni a costo costante, quindi il costo totale è $O(k + m)$.

Intersezioni di semipiani

- Un semipiano è individuato da una disequazione come:

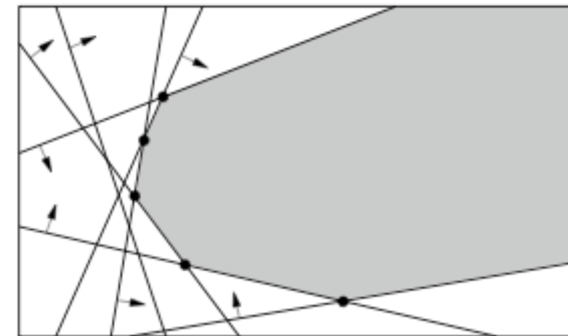
$$h: \quad ax + by + c \leq 0$$

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

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

si tratta di una regione poligonale convessa, non necessariamente limitata.

- ***Problema:***
(HALF-PLANE INTERSECTION) dati n semipiani $h_1 \dots h_n$, produrre lista delle rette che delimitano $h_1 \cap h_2 \cap \dots \cap h_n$ ordinate in senso antiorario.



- Soluzione incrementale. Assumiamo di avere già intersecato $h_1 \cap h_2 \cap \dots \cap h_k$, che formano o una regione poligonale convessa con al più k spigoli. L'intersezione di questa con h_{k+1} si effettua con costo $O(k)$. Dunque il lavoro totale richiesto è $\sum_{k=1}^{n} O(k) = O(n^2)$. Si può fare di meglio ...

Algoritmo "divide-and-conquer"

- L'algoritmo ricorsivo segue lo schema classico del divide-et-impera. Si articola quindi in tre fasi:

  Dividi: Partiziona H in due insiemi H_1 e H_1 di cardinalità $n/2$.

  Domina: Calcola ricorsivamente le intersezioni in H_1 ed in H_2.

  Combina: Interseca le regioni poligonali risultanti.

- Lo schema è come in MergeSort, con il lavoro che si svolge nella fase "combina".

- ***Complessità***: poiché l'intersezione di due regioni poligonali con $n/2$ spigoli ciascuna costa $O(n)$, il costo computazionale dell'algoritmo è dato dalla ricorrenza: $T(n) = 2T(n/2) + O(n)$, la cui soluzione è $O(n \log n)$

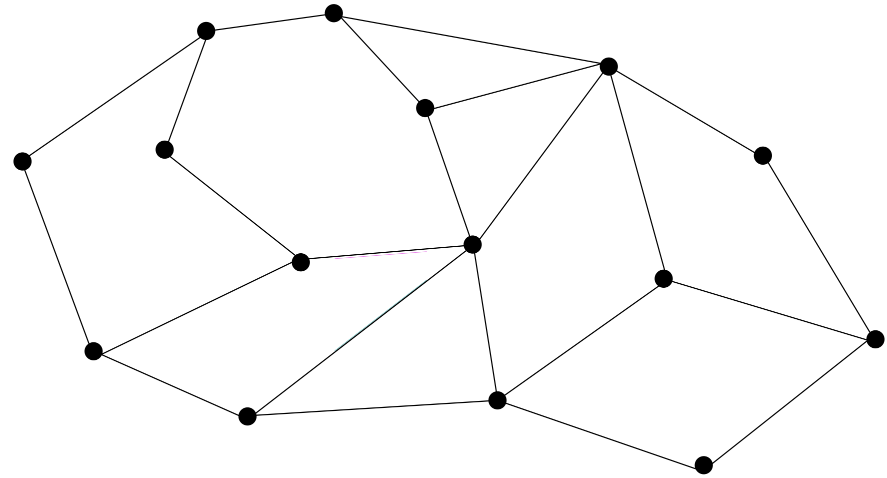
Intersezioni di semispazi (3D)

(cenni)

- ***Problema:*** (HALF-SPACE INTERSECTION) dati n semispazi $h_1 \dots h_n$ in E^3, determinare la loro intersezione: $h_1 \cap h_2 \cap \dots \cap h_n$.
- La soluzione è un poliedro convesso (anche illimitato), 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 "combina") 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).

Suddivisioni piane

- Informalmente,
  una ***suddivisioni piana*** è una partizione di una regione limitata del piano in poligoni semplici.



Una suddivisione piana (non convessa)

- Molte strutture della geometria computazionale sono suddivisioni piane: le triangolazioni, le disposizioni di rette, i diagrammi di Voronoi, le mappe trapezoidali.

- In questo capitolo studieremo i ***grafi piani***, che ci serviranno per definire le suddivisioni piane.

- Descriveremo quindi due delle strutture citate: le ***triangolazioni*** e le ***disposizioni di rette***.

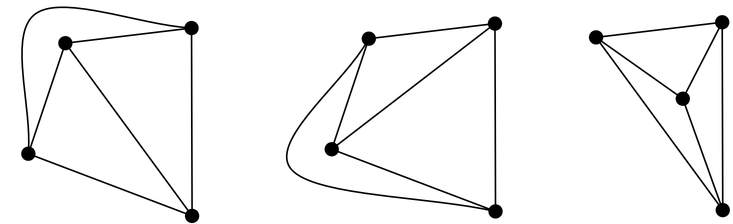
- I diagrammi di Voronoi (assieme alla triangolazione di Delaunay) verranno introdotti più avanti, mentre le mappe trapezoidali non verranno trattate.

Grafi piani

- Un ***grafo*** (non orientato) è una coppia $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ dove $\mathcal{V}$ è un insieme finito non vuoto di elementi chiamati ***vertici*** ed $\mathcal{E}$ è un insieme finito di coppie non ordinate di elementi di $\mathcal{V}$ chiamati ***spigoli***. Uno spigolo $\{A, B\}$ si dice che ***collega*** i vertici A e B.

- Un grafo si dice ***connesso*** se per ogni coppia di vertici esiste una successione di spigoli che li collega.

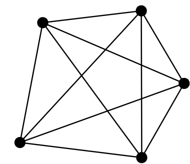
- ***Definizione***. Una ***immersione di un grafo*** nel piano è una particolare rappresentazione del grafo in cui i vertici sono punti del piano e gli spigoli sono segmenti aperti di curva, in modo che ***nessuna coppia di spigoli si intersechi*** (gli estremi non contano perché gli spigoli sono aperti)



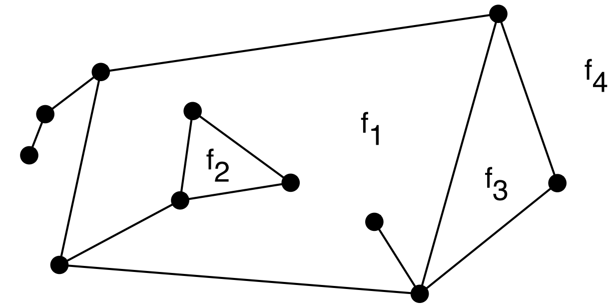
Tre diverse immersioni di un grafo planare.

- Un grafo è ***planare*** se può essere immerso nel piano.

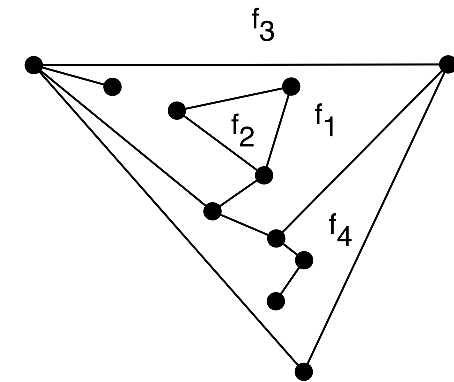
- Il grafo completo con 5 vertici (K_5) non è planare.



- L'immersione di un grafo planare si chiama ***grafo piano***.
- Le ***facce*** di un grafo piano sono le regioni di piano (aperte) delimitate dagli spigoli.
- Più formalmente: le ***facce*** di un grafo piano $\mathcal{G}$ sono gli insiemi connessi massimali (componenti connesse) di $\mathbb{E}^2 \setminus \mathcal{G}$ (insieme ottenuto sottraendo dal piano Euclideo tutti i punti che stanno su vertici o spigoli di $\mathcal{G}$)



- C'è esattamente ***una faccia illimitata*** (chiamata faccia ***esterna***), tutte le altre sono limitate.
- Non c'è nulla di speciale nell'essere la faccia esterna: qualunque faccia può essere scelta per essere la faccia esterna in una particolare immersione.



Una immersione del grafo precedente in cui la faccia esterna diventa f_3.

- ***Incidenza***: Due elementi diversi (vertici, spigoli, facce) tali che uno appartiene alla chiusura dell'altro si dicono incidenti. Quindi:
  - se un vertice appartiene alla chiusura di uno spigolo, i due si dicono incidenti;
  - se uno spigolo appartiene alla chiusura di una faccia i due si dicono incidenti;
  - se un vertice appartiene alla chiusura di una faccia, i due si dicono incidenti.
- Il ***grado*** di un vertice è pari al numero di spigoli incidenti.
- ***Adiacenza***: La relazione di adiacenza è data per elementi dello stesso tipo. Sono adiacenti:
  - due vertici incidenti sul medesimo spigolo,
  - due spigoli che incidono sullo stesso vertice,
  - due facce incidenti sul medesimo spigolo.
- Spesso si considera la chiusura transitiva della relazione di adiacenza (p. es. tutti gli estremi degli spigoli incidenti in P sono adiacenti a P).

- ***Teorema*** (Formula di Eulero) In un grafo piano connesso con n vertici, e spigoli e f facce si ha: $n - e + f = 2$. (dim per induzione su e, v. [5]).
- ***Lemma 1*** (Handshaking). In un grafo piano $\sum_{V \in \mathcal{V}} \deg(V) = 2e$.
- Dim. Ogni vertice distribuisce un contrassegno a ciascuno spigolo incidente. Servono $\sum_{V \in \mathcal{V}} \deg(V)$ contrassegni. Ogni spigolo ha due vertici incidenti e quindi riceve esattamente 2 contrassegni.
- ***Lemma 2***: In un grafo piano connesso con $n \geq 3$ vertici, si ha $3f \leq 2e$.
- Dim. Ogni faccia distribuisce un contrassegno ad ogni spigolo incidente. Servono almeno $3f$ contrassegni, poiché su ogni faccia incidono almeno 3 spigoli. Ciascuno spigolo riceve un contrassegno per ogni faccia su cui incide, quindi al massimo 2. Quindi il numero totale di contrassegni è $\leq 2e$.
- ***Teorema*** (Complessità del grafo piano): Un grafo piano connesso con n vertici ha al più $3(n-2)$ spigoli e $2(n-2)$ facce. (Dim. Sostituire il Lemma 2 nella formula di Eulero.)
- ***Corollario***. Ogni grafo piano contiene (almeno) un vertice di grado 5 o minore. (Dim. Se tutti i vertici avessero almeno 6 spigoli incidenti avremmo una contraddizione per il Teorema precedente).

- ***Corollario***. Il grado ***medio*** di un grafo piano è minore di 6. Infatti applicando il Teorema precedente ed il Lemma 1 si ha:

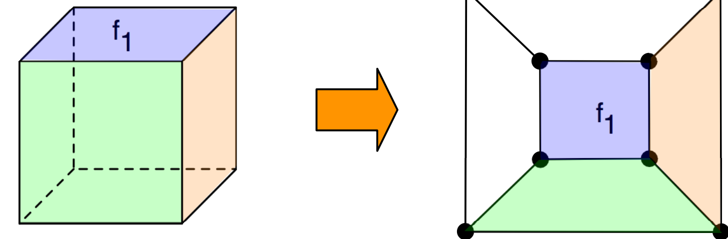
$$\sum_{V \in \mathcal{V}} \deg(V) = 2e \leq 6n - 12 < 6n.$$

- Abbiamo visto che nell'immersione di un grafo sono consentiti spigoli curvilinei, tuttavia:

- ***Teorema*** (Fary): ogni grafo planare ammette una immersione con spigoli rettilinei.

- Un grafo ***piano a spigoli rettilinei*** si chiama PSLG (*plane straight-line graph*)

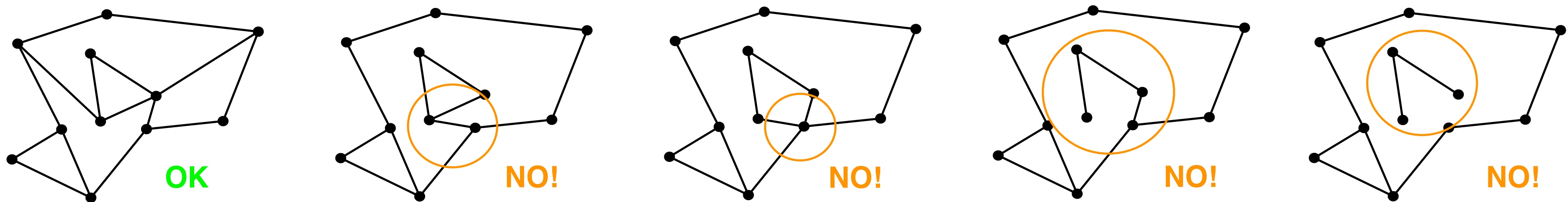
- Nota:
  i ***poliedri*** sono omeomorfi (topologicamente equivalenti) ad un grafo piano (o un PSLG).



- Per trasformare un poliedro convesso in un PSLG lo si proietti prima sulla sua sfera circoscritta e poi su un piano con proiezione stereografica. La faccia in cui giace il punto di proiezione (polo nord) diventa la faccia esterna.

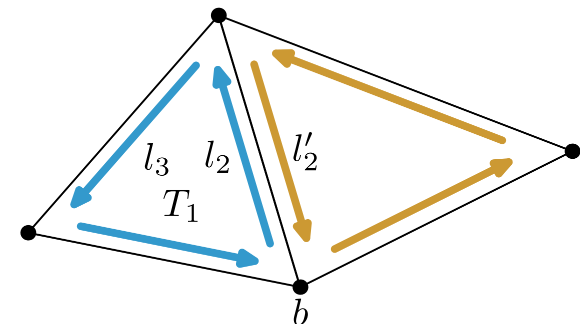
Suddivisione piana

- Un PSLG induce una suddivisione del piano in facce, chiamata ***suddivisione piana***.
- Considereremo suddivisioni piane in cui ***le facce limitate sono poligoni***.
- Per avere questa proprietà (v. [5]) si deve richiedere che
  - il PSLG sia connesso.
  - in ciascun vertice incidano ***esattamente due spigoli*** incidenti sulla stessa faccia.



DCEL

- La DCEL (***Doubly Connected Edge List***) è una struttura dati per la rappresentazione di suddivisioni piane (in principio di PSLG).
- Ha il pregio di mettere in luce tutte le entità geometriche coinvolte (vertici, spigoli e facce) e di agevolare le ricerche di incidenza.
- La versione che descriviamo qui funziona per suddivisioni piane le cui facce limitate sono ***poligoni semplici***.
- Una DCEL consiste di tre tipi di record: vertici, facce e semispigoli. Infatti, uno spigolo si considera scomposto in due ***semispigoli*** orientati in verso opposto.



  - ***Vertice*** Ciascun vertice contiene le coordinate ed il puntatore `v.inc_edge` ad uno dei semispigoli che hanno il vertice come origine.
  - ***Faccia***. La faccia contiene il puntatore ad uno dei semispigoli su cui incide, `f.inc_edge`.

- *Semispigolo*. Il semispigolo e contiene:
  * puntatore al vertice origine, `e.orig`;
  * puntatore al semispigolo gemello, `e.twin`. Basta il solo vertice origine poiché l'altro vertice si recupera come `e.twin.orig`;
  * puntatore alla faccia `e.left` su cui incide. L'orientazione del semispigolo è scelta in modo che la faccia `e.left` giaccia alla sua sinistra;
  * puntatori `e.prev` ed `e.next` al precedente ed al successivo semispigolo del contorno della faccia `e.left`.

```
typedef struct {
   Vertex* orig;
   Edge* twin;
   Face*   left;
   Edge* next;
   Edge* prev;
} Edge;
```

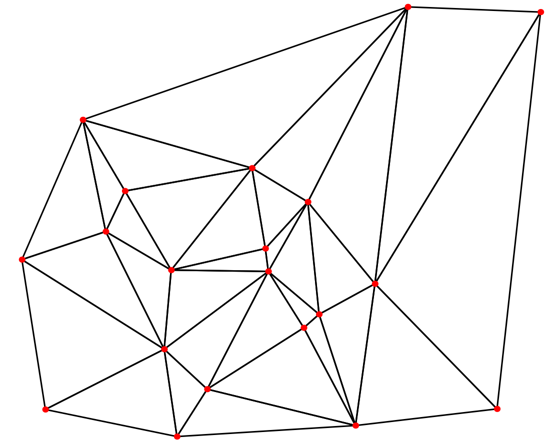
```
typedef struct {
   float x, y, z;
   Edge* inc_edge;
} Vertex;

typedef struct {
   Edge* inc_edge;
} Face;
```

- Le informazioni archiviate in una DCEL consentono di effettuare agevolmente le più comuni operazioni, come: visitare in ordine gli spigoli che contornano una faccia, visitare gli spigoli incidenti su un vertice, etc. ... A titolo di esempio vediamo come enumerare i vertici incidenti su una faccia (dal [6]):

```
enumerate_vertices(Face f) {
    Edge start = f.inc_edge;
    Edge e = start;
    do {
        output e.orig;
        e = e.next;
    } while (e != start);
}
```

- La DCEL si può costruire in tempo lineare a partire da una classica rappresentazione a lista di vertici + lista di adiacenza.

Triangolazione di punti



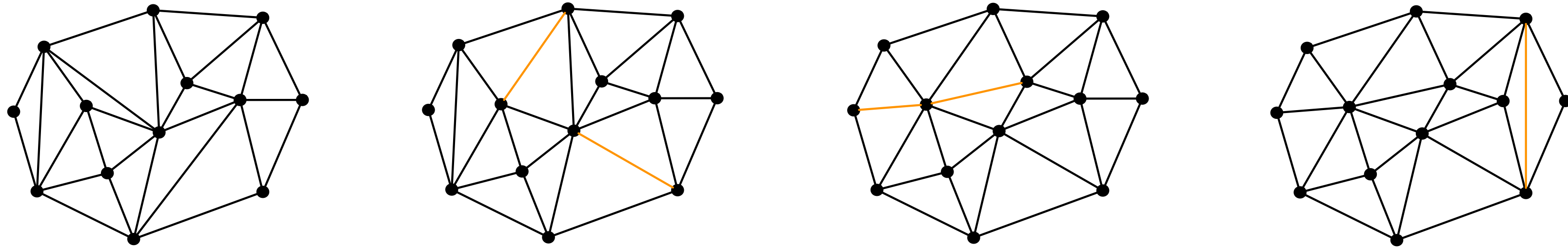
- ***Definizione 1***:
  Una ***triangolazione*** di un insieme finito di punti $\mathcal{S}$ è una suddivisione piana connessa in cui l'insieme dei vertici è $\mathcal{S}$ e ***tutte le facce limitate sono triangoli***.
- Una suddivisione piana $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ è ***massimale*** se non è possibile aggiungere a $\mathcal{E}$ alcuno spigolo senza perdere la planarità (ogni spigolo non in $\mathcal{E}$ interseca uno degli spigoli di $\mathcal{E}$).
- ***Osservazione***: in una suddivisione piana massimale ogni faccia limitata è un triangolo. Infatti ogni faccia limitata è un poligono, e un poligono può sempre essere triangolato.

- ***Definizione 2***: La ***triangolazione*** di un insieme finito di punti $\mathcal{S}$ è una ***suddivisione piana massimale*** il cui insieme di vertici è $\mathcal{S}$.
- ***Osservazione***: L'unione delle facce limitate (triangoli) coincide con il guscio convesso. La faccia esterna è l'insieme complementare del guscio convesso.

- ***Problema:*** (TRIANGULATION) dato un insieme $\mathcal{S}$ di n punti nel piano produrre la descrizione di una loro triangolazione.

- La descrizione può essere, p.es., una DCEL.

- È facile rendersi conto con esempi che un insieme di punti ammette più di una triangolazione.

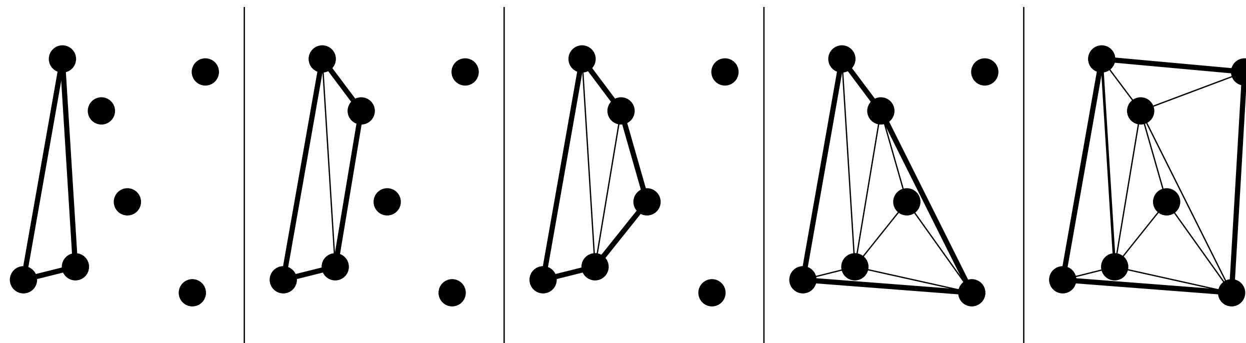


- Si può rendere più vincolato il problema richiedendo qualche proprietà particolare alla triangolazione, come vedremo più avanti con la triangolazione di Delaunay.

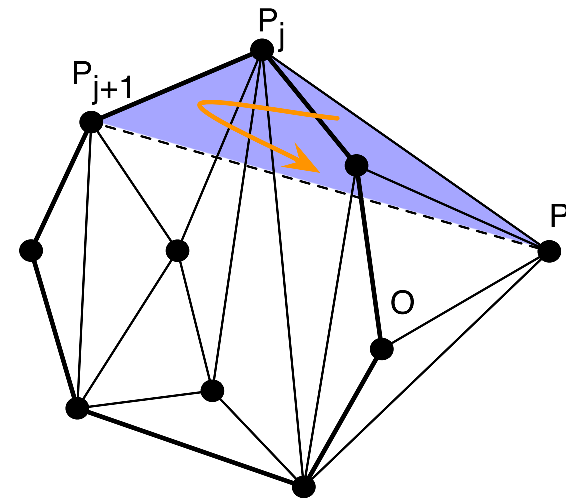
Algoritmo incrementale

- ***Risolve*** TRIANGULATION.
- Deriva direttamente dall'algoritmo incrementale per il guscio convesso. La presentazione sarà comunque autosufficiente. Lo schema generale è il seguente:

1. Si ordinano i punti di $\mathcal{S}$ secondo l'ordine lessicografico su (x, y).
2. Si costruisce il triangolo formato dai primi tre punti (che coincide anche con il loro guscio convesso).
3. Si aggiungono uno alla volta i punti successivi: l'inserimento del punto P consiste nel collegarlo (formando un nuovo spigolo della triangolazione) a ciascun vertice P_j appartenente alla triangolazione corrente, che sia ***visibile*** da P (ovvero tale che il segmento $\overline{PP_j}$ non intersechi la triangolazione).



- Sia $\mathcal{T}$ la triangolazione corrente e $\mathcal{Q}$ il corrispondente guscio convesso.
- Per effettuare in modo efficiente l'aggiunta del punto P (evitando di controllare tutti i vertici della triangolazione), bisogna osservare che:
  - In virtù dell'ordinamento lessicografico, P è esterno a $\mathcal{Q}$, ed il punto O che precede P nell'ordinamento (l'ultimo inserito) è sempre visibile da P.
  - Gli spigoli di $\mathcal{Q}$ fanno parte di $\mathcal{T}$, quindi i punti di $\mathcal{T}$ visibili da P devono appartenere a $\mathcal{Q}$, poiché quelli interni sono sicuramente non visibili.
- ***Idea***: muoversi lungo $\mathcal{Q}$ a partire da O verso l'alto e verso il basso ed unire i punti incontrati a P con uno spigolo (che viene aggiunto a $\mathcal{T}$) finché non si incontra un punto non visibile. Gli ultimi punti visibili incontrati prendono il nome di punti di ***tangenza*** (sono due, superiore ed inferiore).



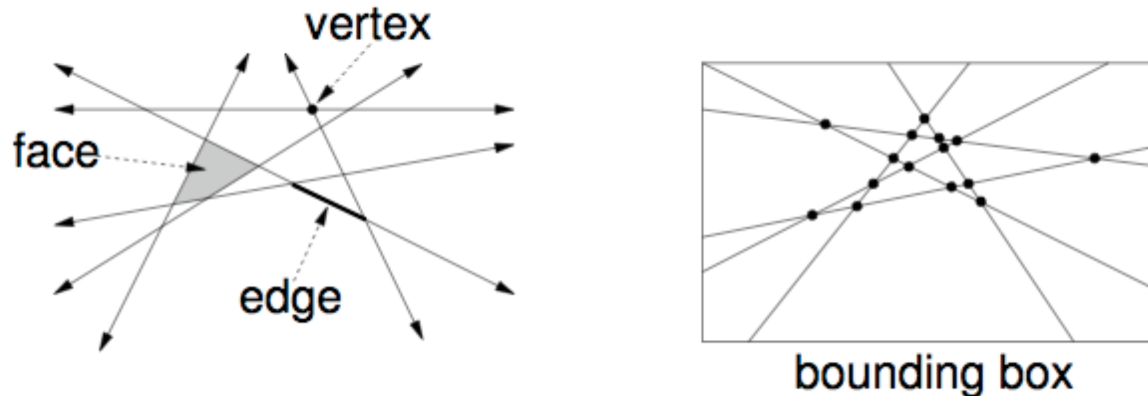
- Il test di visibilità (nella parte superiore della scansione) si effettua controllando orient(P, P_j, P_{j+1}), dove dove P_j è un punto di $\mathcal{Q}$ e P_{j+1} è il suo successore (in senso antiorario) in $\mathcal{Q}$. Se i tre punti definiscono una svolta antioraria, allora P_j è un punto di tangenza.
- Il test è analogo (mutatis mutandis) per la parte inferiore.

- Quando si aggiorna $\mathcal{T}$ in seguito alla introduzione di un punto, bisogna anche aggiornare $\mathcal{Q}$, rimuovendo tutti i punti visibili da P esclusi i due punti di tangenza, e inserendo P tra i due.

- ***Complessità***: I vertici, che sono n, vengono inseriti uno alla volta. Ad ogni passo vengono creati un certo numero k di spigoli e nessuno spigolo viene mai tolto dalla triangolazione. Il costo per inserire k spigoli consiste nel visitare $k+2$ vertici, dunque il costo per spigolo è $O(1)$. In totale gli spigoli inseriti sono $O(n)$ (in un PSLG il numero di spigoli è $O(n)$). Domina quindi l'ordinamento dei punti, ed il costo totale è $O(n \log n)$.

Disposizione di rette

- Un insieme finito $\mathcal{L}$ di rette nel piano induce una partizione del piano in insiemi poligonali (limitati e non), che prende il nome di ***disposizione*** $\mathrm{D}(\mathcal{L})$.



- La disposizione non è tecnicamente una suddivisione piana a causa degli spigoli illimitati.
- Possiamo però racchiudere la parte "interessante" della disposizione in un rettangolo e tagliare gli spigoli che escono.
- In questo modo ci siamo ricondotti ad una suddivisione piana (che possiamo rappresentare con una DCEL).
- Possiamo dunque definire vertici, spigoli e facce della disposizione in modo naturale.

- Assumiamo che la disposizione sia semplice, ovvero che non esitano tre rette che si intersecano in un punto e che non vi siano rette parallele.

- Teorema: Dato un insieme finito $\mathcal{L}$ di rette nel piano:

  i) il numero di vertici di $D(\mathcal{L})$ è $\binom{n}{2}$

  ii) il numero di spigoli di $D(\mathcal{L})$ è n^2

  iii) il numero di facce di $D(\mathcal{L})$ è $\binom{n}{2} + n + 1$

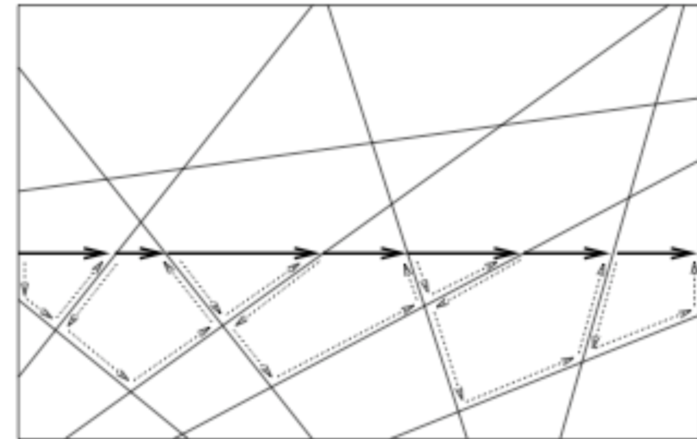
  (dim. sul [6]).

  Problema: (Lines Arrangement): Dato un insieme $\mathcal{L}$ di n rette, produrre una descrizione della suddivisione piana corrispondente.

- Il primo algoritmo che viene in mente è il *plane sweep* che abbiamo usato per il Line-Segment Intersection. Infatti è facile modificarlo per questo scopo, ma poiché il numero di punti di intersezione è $O(n^2)$, l'algoritmo richiederebbe $O(n^2 \log n)$. Non male, ma si può fare di meglio...

Algoritmo incrementale

- ***Risolve*** LINES ARRANGEMENT.
- Si aggiunge una retta alla volta, senza un particolare ordine (costruzione incrementale).
- L'aggiunta di una retta ℓ_i alla disposizione $D(\{\ell_1, \dots \ell_{i-1}\})$ consiste nel suddividere le facce attraversate da ℓ_i. Queste si possono determinare attraversando ℓ_i da sinistra a destra e cercando le intersezioni con gli spigoli di $D(\{\ell_1, \dots \ell_{i-1}\})$.



- Ogni volta che si entra in una faccia, bisogna determinare il punto di uscita, ovvero quale altro spigolo della faccia interseca ℓ_i.

- In dettaglio, supponiamo che la disposizione sia rappresentata da una struttura dati (come DCEL) che consenta la visita degli spigoli di una faccia e la visita di facce adiacenti.
  - Ogni volta che si entra in una faccia si visitano gli spigoli della faccia in senso antiorario a partire dallo spigolo di ingresso, effettuando il test di intersezione.
  - Quando si trova lo spigolo di uscita si passa alla faccia incidente sullo stesso spigolo e si continua.
  - Al termine si aggiorna la struttura dati aggiungendo uno spigolo per ciascuna coppia di punti di ingresso-uscita dalla faccia.
- ***Complessità***: Se l'inserimento di ℓ_i avviene in $O(i)$, la complessità totale è $O(n^2)$. La dimostrazione che l'inserimento di una retta avviene in tempo lineare si basa sul seguente:
- ***Teorema*** (zone theorem) Data una disposizione $D(\mathcal{L})$ di n rette nel piano, e data una retta ℓ, il numero totale degli spigoli delle facce la cui chiusura interseca ℓ è al più $6n$. (v. [2] e [6]).

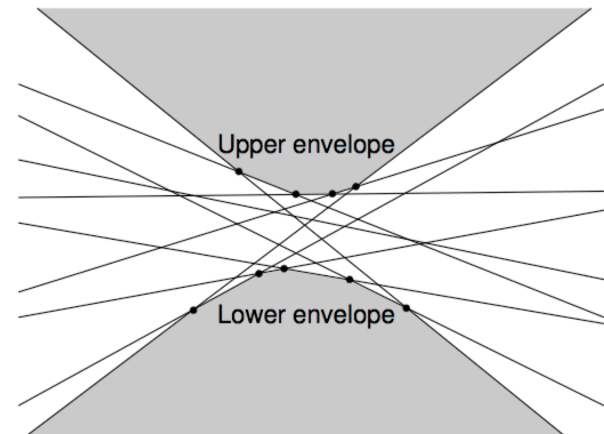
Dualità punto-retta

- Contrariamente ad altre strutture notevoli delle Geometria Computazionale, che sono definite in termini di punti, la disposizione di rette coinvolge un insieme di rette (!!).
- Tuttavia le disposizioni di rette sono spesso impiegate per risolvere problemi su insiemi di punti, tramite la ***dualità punto-retta***.
- La dualità punto-retta è una mappa che trasforma punti in rette e viceversa.
- A volte è possibile prendere un problema definito sui punti e mapparlo su un problema equivalente definito sulle rette, e viceversa.
- Spesso questo processo migliora la comprensione del problema e/o rende evidenti strutture e relazioni tra le entità. Oppure semplicemente ci risparmia l'ideazione di un nuovo algoritmo.

- Una retta nel piano primale (x, y) corrisponde ad un punto nello spazio duale secondo la mappa: $\ell : y = ax - b \leftrightarrow \ell^* = P : (a, b)$
- Un punto (P_x, P_y) nel piano primale è individuato da un fascio di rette di equazione: $P_y = aP_x - b$. Tali rette corrispondono ad un insieme di punti $\mathcal{L}$ nel piano duale: $\mathcal{L} = \{(a, b) | P_x = aP_y - b\} = \{(a, b) | b = P_x a - P_y\}$. Tale insieme è una retta nel piano duale che ha coefficiente angolare P_x e intercetta le ordinate in $-P_y$. Dunque:
- Un punto nel piano primale corrisponde ad una retta nel piano duale secondo la mappa: $P : (x, y) \leftrightarrow P^* = \ell : b = xa - y$.
- Alcune proprietà (si verificano analiticamente):
  - ***Idempotenza:*** $(P^*)^* = P$
  - ***Inversione dell'ordine:*** Il punto P giace sopra/sotto la retta ℓ nel piano primale se e solo se la retta P^* passa sotto/sopra il punto ℓ^*.
  - ***Intersezione:*** Le rette ℓ_1 ed ℓ_2 nel piano primale si intersecano in un punto P se e solo se la retta P^* passa per i punti ℓ_1^* ed ℓ_2^* nel piano duale.
  - ***Collinerità:*** Tre punto sono collineari nel piano primale se e solo se le loro rispettive rette duali si intersecano in un punto comune.

- ***Esempio1***: si consideri il seguente problema:
- General Position Test: dati n punti nel piano determinare se ve ne sono almeno 3 collineari.
- L'algoritmo che controlla ogni terna di punti impiega $O(n^3)$, essendo $\binom{n}{3}$ le terne.
- Se invece si considera il piano ***duale***, si tratta di controllare se esiste un vertice di grado maggiore di 4, con un costo di $O(n^2)$.

- ***Esempio 2***: Half-plane Intersection è il ***duale*** di Convex Hull, secondo i termini che ora preciseremo meglio.
- L'***inviluppo superiore (inferiore)*** di un insieme di n rette $\mathcal{L}$ è l'intersezione dei semipiani che giacciono sopra (sotto) le rette date.
- ***Problema:*** (Upper/Lower envelope) Date n rette non orizzontali si determini l'intersezione dei semipiani che giacciono sopra (sotto) le rette.



- Se si escludono le rette verticali, si può risolvere HALF-PLANE INTERSECTION con UPPER/LOWER ENVELOPE: basta infatti partizionare i semipiani originali in quelli che giacciono sopra e quelli che giacciono sotto la retta di supporto, risolvere rispettivamente UPPER ENVELOPE e LOWER ENVELOPE ed intersecare i risultati.

- ***Proposizione:*** Sia $\mathcal{S}$ un insieme di punti nel piano. Il guscio (convesso) superiore di $\mathcal{S}$ coincide con l'inviluppo inferiore del duale $\mathcal{S}^*$.

- Dim. Due punti P e Q di $\mathcal{S}$ sono uno spigolo del guscio superiore se e solo se tutti gli altri punti giacciono al di sotto della retta ℓ_{ij} passante per P e Q. Nel piano duale questo si traduce in: il punto di intersezione ℓ_{ij}^* delle rette P^* e Q^* lascia tutte la altre rette di $\mathcal{S}^*$ al di sopra, dunque ℓ_{ij}^* è un vertice dell'inviluppo inferiore e P^* e Q^* sono adiacenti. Dunque la sequenza degli spigoli del guscio superiore coincide con la sequenza dei vertici dell'inviluppo inferiore.

Ricerca Geometrica

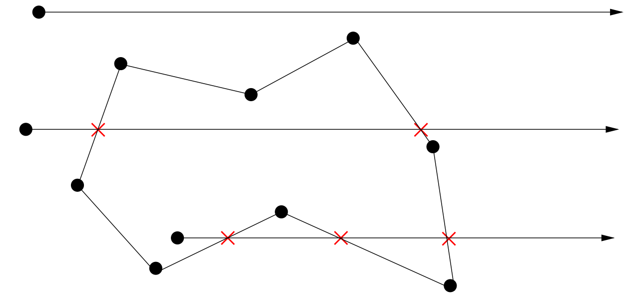
- Supponiamo di avere dei dati accumulati ("archivio") ed un dato nuovo ("campione"). La ricerca geometrica consiste nel collegare il campione con l'archivio.
- Vi sono fondamentalmente due paradigmi di ricerca geometrica:
  1. ***Problemi di localizzazione***, dove l'archivio rappresenta una partizione del piano 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 una regione del piano (tipicamente un rettangolo o una circonferenza). Il *range search* consiste nell'elencare (*report problem*) o contare (*count problem*) tutti i punti contenuti nella regione data.
- Se la ricerca non viene eseguita una volta sola (*single-shot*), ma si pensa che si presenteranno più interrogazioni sugli stessi dati (*repetitive-mode*), allora vale la pena di investire tempo e spazio nel pre-elaborarli in modo opportuno per effettuare poi la ricerca più velocemente.

Localizzazione di un punto in un poligono

- Il caso più semplice di partizione del piano è quello in cui il piano è suddiviso in due regioni: l'interno di un poligono semplice e l'esterno.
- ***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, si chiede se Q giace all'interno o all'esterno del poligono.
- ***Problema 2:*** (CONVEX POLYGON INCLUSION) Dato un poligono convesso e dato un punto Q, si chiede se Q giace all'interno o all'esterno del poligono.

Metodo di Jordan

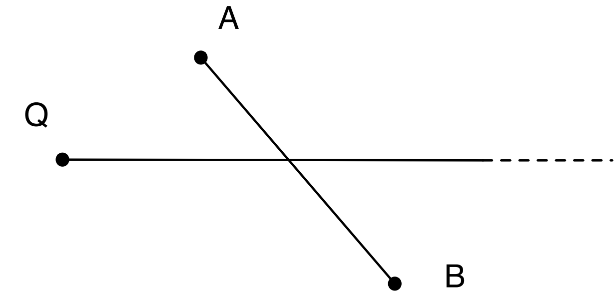
- ***Risolve*** POLYGON INCLUSION
- L'algoritmo ricalca la dimostrazione di Jordan.
- Consideriamo la semiretta orizzontale con origine in Q e diretta verso destra. Il punto all'infinito è per definizione all'esterno del poligono.



- 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.

- Vediamo come effettuare il ***test di intersezione semiretta-segmento***: data una semiretta orizzontale ℓ da $-\infty$ a Q ed un segmento $\overline{AB}$ dire se ℓ interseca $\overline{AB}$.

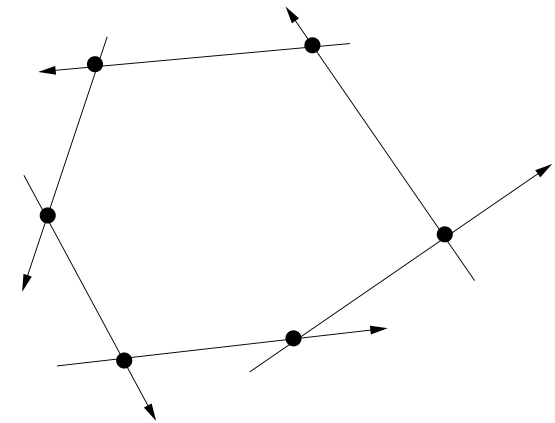
- Se il segmento è orizzontale c'è intersezione solo se il segmento è contenuto nella semiretta. Se $\overline{AB}$ non è orizzontale, supponiamo che sia $A_y > B_y$. C'è intersezione se e solo se:



$$B_y \leq Q_y \leq A_y \quad \text{e} \quad \text{orient}(A, B, Q) < 0$$

- ***Complessità***: devo effettuare il controllo di intersezione tra la semiretta e ciascuno spigolo del poligono. Il test ha costo costante, quindi $O(n)$.

Metodo dei semipiani

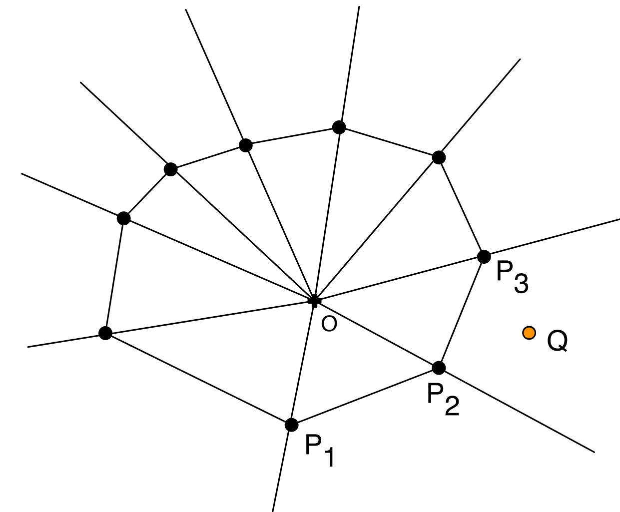


- ***Risolve*** CONVEX POLYGON INCLUSION
- Per quanto il metodo di Jordan si possa applicare anche a poligono convessi, esiste un metodo specifico per questi ultimi di più facile implementazione (ma la complessità è la stessa).
- L'interno di un poligono convesso è l'intersezione dei semipiani aperti che contengono i suoi spigoli sulla frontiera.
- Quindi: Q giace all'interno del poligono se e solo se Q giace a sinistra di tutte le rette orientate passanti per gli spigoli del poligono (la retta orientata da P_i a P_{i+1} $\forall i = 1 \dots n$)

- Q giace sulla frontiera del poligono se Q è allineato con almeno una delle rette sopra menzionate, e giace a sinistra di tutte le altre.
- Basta usare $\text{orient}(P_{i+1}, P_i, Q)$ per scoprire da che parte giace Q rispetto alla retta.
- ***Complessità***: Il test ha costo costante. Nel caso peggiore (punto interno) devo controllare tutti i vertici, quindi $O(n)$

Metodo delle fette

- Vediamo un metodo con pre-elaborazione che risponde in $O(\log n)$.
- Il fatto che i vertici di un poligono convesso siano ordinati per coordinata polare (rispetto ad un qualunque punto interno), suggerisce immediatamente la possibilità di una ricerca dicotomica.
- Si prende un punto interno O qualunque e si tracciano le semirette che hanno origine in O e passano per i vertici del poligono. Le semirette partizionano il piano in fette, e ciascuna fetta è divisa in due regioni dallo spigolo del poligono: una è interna al poligono, l'altra è esterna.



- La fetta cui appartiene il punto di interrogazione Q si trova con ricerca dicotomica, visto che le semirette che delimitano le fette sono ordinate. Poi si controlla da che parte giace Q rispetto allo spigolo che appartiene alla fetta.
- ***Complessità***: la risposta viene fornita in $O(\log n)$. La pre-elaborazione consiste nel determinare il punto interno O. Si può prendere il baricentro dei primi 3 vertici non collineari che si incontrano. Questo è $O(n)$ nel caso pessimo, ma in pratica è $O(1)$.

- Variante: prendo come punto O il punto all'infinito, ottenendo fette orizzontali.

Localizzazione di un punto in un poliedro (3D)

(cenni)

- Per risolvere CONVEX POLYHEDRON INCLUSION si usa la definizione di poliedro convesso come intersezione dei semispazi individuati dalle facce, e quindi si usa $\text{orient}(P, Q, R, S)$ per controllare se il punto da localizzare si trova a sinistra di tutte le facce del poliedro.

- Per risolvere 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 (che sono poligoni).

- L'intersezione retta-poligono in 3D si effettua in due passi:
  1. Si controlla se la retta interseca il piano contente il poligono ed eventualmente si calcola il punto di intersezione P.
  2. Si localizza P rispetto al poligono nel piano.

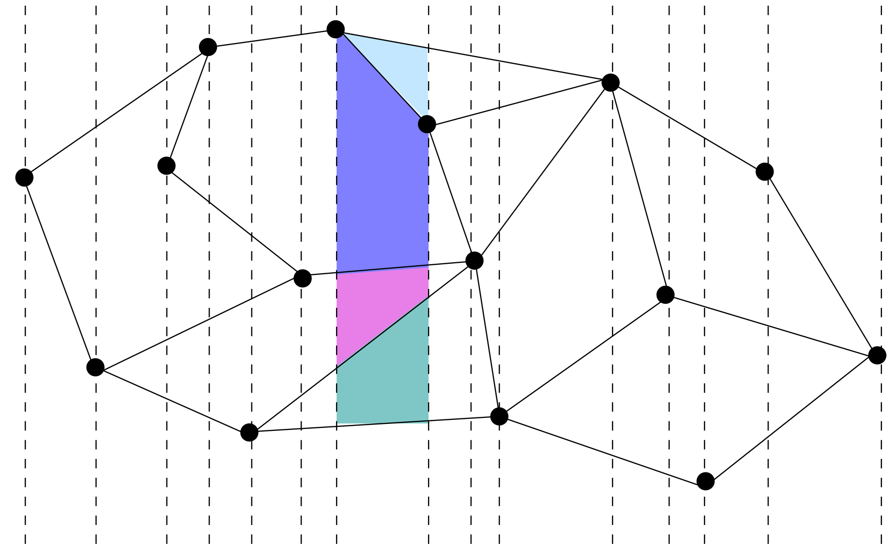
Localizzazione di un punto in una suddivisione piana

- ***Problema:*** (Planar Point Location) Data una ***suddivisione piana*** $\mathcal{G}$ con n vertici e dato un punto Q, trovare la faccia della suddivisione che contiene Q.
- Visto che le facce della suddivisione sono poligoni semplici – in generale non convessi – viene subito in mente un metodo diretto basato sulla localizzazione di un punto in un poligono:
  - si scorrono tutte le facce della suddivisione
  - per ciascuna faccia (poligono semplice) si risolve Polygon Inclusion.
- La struttura dati per la suddivisione deve consentire di ricavare i vertici incidenti su una faccia in tempo lineare.
- ***Complessità***: lineare nel numero dei vertici: $O(n)$
- Anche in questo caso vediamo se con una opportuna pre-elaborazione è possibile ottenere costo $O(\log n)$ tramite ricerca dicotomica.

Metodo delle strisce (Slab method)

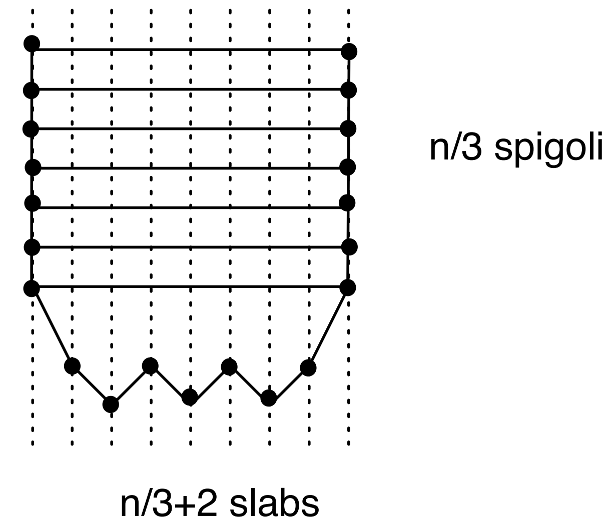
(Dobkin e Lipton, 1976)

- ***Risolve*** Planar Point Location
- ***Idea***: sovrapporre alla suddivisione piana $\mathcal{G}$ una struttura che consenta la ricerca dicotomica.
- Si suddivide il piano in strisce verticali passanti per i vertici di $\mathcal{S}$ (ce ne sono $n-1$).



- Poiché gli spigoli si intersecano solo in corrispondenza dei vertici, essi non si intersecano all'interno delle strisce (per costruzione).
- Dunque i segmenti all'interno di ciascuna striscia possono essere ordinati, per esempio dal basso verso l'alto.

- L'intersezione di una striscia con un poligono della suddivisione è un ***trapezoide***.
- Per localizzare un punto Q si compie una ricerca sulle ascisse per individuare la striscia contenente la Q_x, seguita da una ricerca sulle ordinate per individuare il trapezoide che contiene la Q_y.
- ***Complessità***: Le strisce sono al più $n-1$, quindi la ricerca (dicotomica) sulle x costa $O(\log n)$. Gli spigoli che intersecano una striscia sono $O(n)$, quindi anche la ricerca dicotomica sulle y costa $O(\log n)$.
- C' è però da tenere presente anche il costo di pre-elaborazione per costruire la struttura dati spaziale che rappresenta il contenuto delle strisce (ovvero i segmenti ordinati per y), e lo spazio occupato.
- Per quanto riguarda lo spazio, notiamo subito che nel caso peggiore serve $O(n^2)$ spazio, poiché esistono casi in cui le strisce contengono complessivamente $O(n^2)$ segmenti



Un caso in cui ci sono $O(n^2)$ segmenti nelle strisce.

- Per quanto riguarda il ***tempo di pre-elaborazione***, una approccio naive richiederebbe $O(n^2 \log n)$: infatti, in ciascuna striscia ci sono $O(n)$ segmenti che vengono ordinati, al costo di $O(n \log n)$, e ci sono in tutto $O(n)$ strisce.

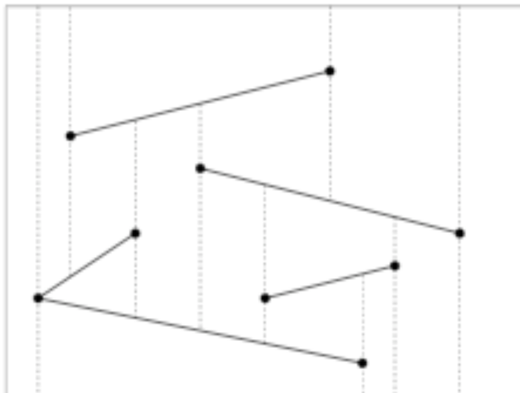
- Questo modo di procedere, tuttavia, non sfrutta la coerenza spaziale tra strisce contigue, che suggerisce che si possa ottenere il contenuto di una striscia modificando localmente quello della striscia precedente.

- In effetti il contenuto delle striscia può essere efficientemente costruito sfruttando l'algoritmo ***plane sweep*** di Bentley-Ottmann visto precedentemente, con eventi tutti noti a priori (la coda degli eventi può essere un vettore).

- In corrispondenza di una striscia, l'ordine dei segmenti al suo interno è ottenuto leggendo lo stato della *sweep line* (un albero bilanciato) quando questa è all'interno della striscia.

- La complessità temporale della spazzolata, come nell'algoritmo originale, è $O(n \log n)$ (un grafo planare ha $O(n)$ spigoli) mentre la generazione dell'output (la copiatura) costa $O(n^2)$ poiché, come osservato, i segmenti possono essere $O(n^2)$ nel caso peggiore. Quindi il costo totale per la pre-elaborazione è $O(n^2)$.

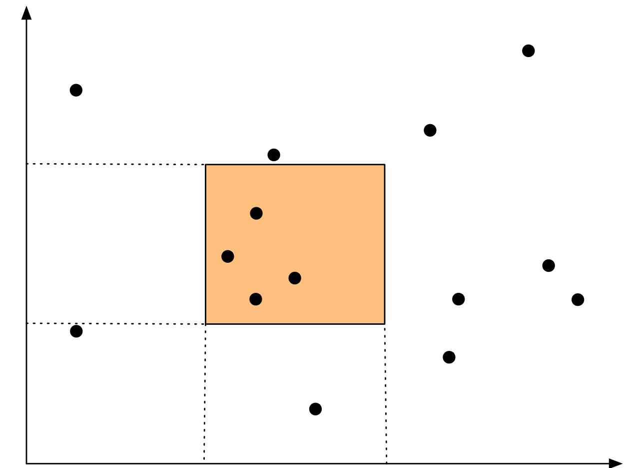
- ***Vantaggi e svantaggi***: il tempo $O(\log n)$ per rispondere all'interrogazione dell'algoritmo delle strisce è ottimo. Tuttavia questo avviene a spese di una occupazione e tempo di pre-elaborazione quadratici, che è inaccettabile in certe applicazioni.
- Esistono metodi che raggiungono spazio $O(n)$ e tempo di pre-elaborazione $O(n \log n)$.
- In [6] sono descritti l'algoritmo di Kirkpatrick ed il metodo della mappa trapezoidale (trapezoidal map).



Mappa trapezoidale

Range search

- ***Problema:*** (RANGE SEARCH - REPORT) Dato un insieme di punti $\mathcal{S}$ in E^d e data una regione R dello spazio (interrogazione), elencare i punti di $\mathcal{S}$ che appartengono ad R.
- Il più semplice problema RANGE SEARCH - COUNT prevede solo di contare il numero di punti che appartengono ad R.
- Ci occuperemo inoltre del caso in cui la regione di ricerca sia un rettangolo con gli spigoli paralleli agli assi cartesiani; tale ricerca prende anche il nome di ***ricerca ortogonale***.
- È chiaro che in modo ***single-shot*** si può effettuare la ricerca in tempo lineare $O(n)$, e questo è ottimale, poiché la regione di ricerca può includere tutti i punti.

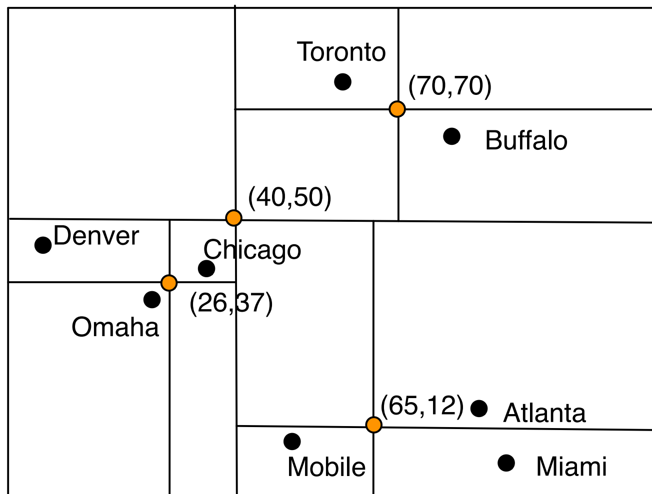


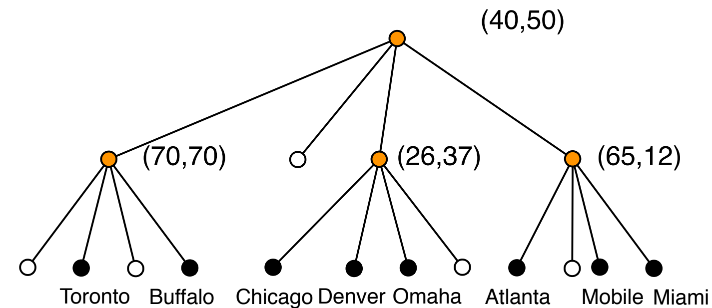
- La nostra attenzione si sposta quindi sul ***repetitive-mode***: introdurremo delle strutture dati che imponendo una organizzazione spaziale gerarchica ai punti renderanno più veloce la ricerca.
- Si noti che – nel caso pessimo – anche l'impiego di una struttura dati non può fare meglio di $O(n)$.
- Solo un analisi *output sensitive* permette di apprezzare il vantaggio apportato dalla struttura dati.
- Tale vantaggio viene pagato in termini di ***spazio*** occupato dalla struttura dati e ***tempo*** speso per la sua costruzione (pre-elaborazione).

Point Quadtree

- Un modo che viene subito in mente per imporre una organizzazione spaziale gerarchica ai punti è suddividere il piano ricorsivamente con rette parallele agli assi, costruendo nel contempo un albero (bilanciato).

- I ***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.

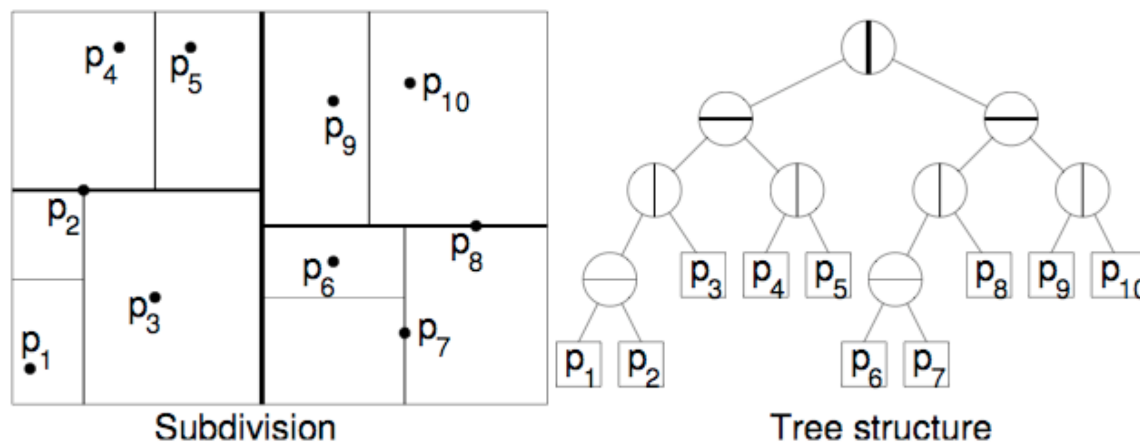




- La versione di quadtree che descriviamo (chiamata pseudo-quadtree in [4]) contiene in punti originali nelle foglie, mentre i punti contenuti nei nodi interni sono fittizi e servono solo per la suddivisione. Ad ogni livello questi vengono scelti in modo da dividere i punti rimanenti in quattro parti di eguale numerosità.
- Il difetto dei quadtree 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.
- Non approfondiamo quindi l'analisi del quad-tree e passiamo ad una loro evoluzione, i ***k-d tree*** che usano una suddivisione binaria invece che quaternaria.

K-d Tree

- Il ***k-d tree*** è simile al quadtree, ma le suddivisioni avvengono lungo una sola retta orizzontale o verticale.
- In ciascun nodo interno è definita una retta, orizzontale o verticale che partiziona i punti, detta ***retta discriminante***.
- Il sottoalbero sinistro di un nodo interno contiene tutti i punti a sinistra/sotto della retta, il sottoalbero destro quelli a destra/sopra.
- Ovvero, se nel nodo corrente la ***coordinata discriminante*** è la x, tutti i punti la cui coordinata x è minore o uguale ad un certo valore (***valore discriminante***) vengono archiviati nel sottoalbero sinistro, gli altri punti nel sottoalbero destro.



- I punti sono memorizzati nelle foglie. Il k-d tree originale invece contiene i punti nei nodi interni. Quella che descriviamo (seguendo [2] e [6]) è una variante chiamata ***adaptive k-d tree*** in [4].

- Il processo di suddivisione assegna ***implicitamente*** una regione del piano rettangolare (***cella***) $\text{reg}(\nu)$ a ciascun nodo interno ν.

- In generale queste celle rettangolari possono essere illimitate, ma di solito ci si restringe ad una regione rettangolare del piano contenente tutti i punti. In questo modo tutte le celle sono limitate.

- Osserviamo che
  - $\text{reg}(\nu)$ è delimitata dalle rette discriminanti associate agli antenati del nodo ν.
  - La retta associata al nodo ν partiziona $\text{reg}(\nu)$.

- Le celle sono annidate, nel senso che le celle dei figli sono contenute nella cella del padre. Definiscono quindi una ***suddivisione gerarchica dello spazio***.

- Ci sono due punti da definire: come scegliere la coordinata discriminante (o ***cutting dimension***), e come scegliere il valore discriminante (o ***cutting value***) in ciascun nodo interno.

- Per quanto riguarda la ***coordinata discriminante***, il metodo più semplice è quello di ***alternare*** su x e y. Ha il vantaggio che non serve archiviare la coordinata discriminante nel nodo, ma essendo indipendente dalla distribuzione dei punti, può produrre celle troppo "snelle".

- Il metodo adattativo prevede di discriminare lungo la dimensione lungo cui i punti sono maggiormente dispersi. La dispersione si calcola (p. es.) come differenza tra la maggiore e la minore coordinata.

- Per quanto riguarda il ***valore discriminante***, per garantire che l'albero sia bilanciato, ovvero che abbia altezza $O(\log n)$, si deve scegliere un valore che divida a metà i punti rimanenti. Il ***valore mediano*** della coordinata discriminante dei punti produce questo effetto.

- Con questa strategia esiste un punto la cui coordinata è uguale al valore discriminante. Usiamo la convenzione di mettere tale punto nel sottoalbero sinistro.

- ***Costruzione del k-d tree***. Procedura ricorsiva:

```
KDNode insert(PointSet P) {
t = new KDNode;
if (#P = 1) /* create a leaf */
     t.point = unique point in P;
     t.left = NIL;  t.right = NIL;
else /* create an internal node */
     t.cutDim =  choose cut dimension;
     t.cutVal =  median(P, t.cutDim);
     t.size = #P;
     split P in P1 and P2  with t.cutDim and t.cutVal;
     t.left = insert(P1);  t.right = insert(P2);
return t
}
```

- ***Tempo per costruzione***: Ad ogni nodo il costo del calcolo della mediana è lineare nel numero dei punti nel sottoalbero, quindi il costo è dato dalla ricorrenza: $T(n) = 2T(n/2) + n$, la cui soluzione è $O(n \log n)$.

- ***Spazio occupato***: $O(n)$, perché i nodi interni di un albero binario con n foglie sono $n - 1$.

- ***Interrogazione del k-d tree***. Vediamo ora come il k-d tree possa servire a rispondere ad interrogazioni di ricerca ortogonale, ovvero a risolvere il problema RANGE SEARCH - REPORT (e RANGE SEARCH - COUNT).
- Si abbia quindi un insieme $\mathcal{S}$ di punti in due dimensioni, archiviati in un k-d tree, e sia $\mathcal{R}$ il rettangolo interrogazione (*query*).
- ***Osservazione*** Un punto è archiviato in una foglia del sottoalbero di ν se e solo se appartiene a $\text{reg}(\nu)$.
- Quindi la ricerca procede ricorsivamente, attraversando l'albero a partire dalla radice, tenendo presente che si deve visitare il sottoalbero che ha per radice ν solo se $\text{reg}(\nu)$ interseca $\mathcal{R}$.
- Più in dettaglio, la visita del nodo ν consiste delle seguenti operazioni:
  1. Se ν è una foglia ed il punto archiviato $\in \mathcal{R}$, allora ritorna il punto;
  2. Se ν è un nodo interno e $\text{reg}(\nu) \subset \mathcal{R}$ allora ritorna (o conta) tutte le foglie del sottoalbero di ν;
  3. Altrimenti, se $\text{reg}(\nu) \cap \mathcal{R} \neq \emptyset$ visita ricorsivamente i sottoalberi destro e sinistro.

- Il seguente pseudo-codice illustra meglio la procedura per rispondere ad una interrogazione Range Search - Count:

```
int rangeCount(Range R, KDNode t, Rectangle C )
 if (t is a leaf)
    if (R contains t.point) return 1,
    else return 0.
 if (t is not a leaf)
    if (C intersection R = emptySet) return 0.
    else if (C contained in R) return t.size.
    else
      split C in C1 and C2  with t.cutDim and t.cutVal
       return (rangeCount(R,t.left,C1) + rangeCount(R,t.right,C2)).
```

- Per quanto riguarda il caso Range Search - Report, l'istruzione `return t.size` viene sostituita con la chiamata alla procedura `ReportSubtree(t)` che visita il sottoalbero che ha il nodo `t` per radice e riporta tutti i punti archiviati nelle foglie.

- ***Complessità della interrogazione***. Consideriamo prima il caso della interrogazione RANGE SEARCH - COUNT. Il costo temporale dipende dal numero di nodi che vengono visitati. La chiamata ricorsiva viene fatta in corrispondenza dei nodi le cui celle intersecano l'interrogazione $\mathcal{R}$ senza però esserne completamente contenute. Diciamo che una tale cella è ***punta*** dalla interrogazione. Si vuole trovare una delimitazione superiore al numero di celle punte da una qualunque interrogazione.

- Consideriamo prima il caso di una interrogazione non rettangolare, data da un semipiano orizzontale o verticale. Le celle punte dal semipiano sono quelle intersecate dalla sua retta di supporto. Si dimostra il seguente:

- ***Lemma***. In un k-d tree bilanciato con dimensione discriminante alternata, una qualunque retta orizzontale o verticale interseca $O(\sqrt{n})$ celle. (dim. sul [6])

- L'interrogazione rettangolare è l'intersezione di 4 semipiani, dunque un rettangolo punge al più $O(4\sqrt{n}) = O(\sqrt{n})$ celle. Concludiamo che vengono visitati $O(\sqrt{n})$ nodi.

- Per quanto riguarda il caso Range Search - Report, la chiamata alla procedura `ReportSubtree(t)` ha costo lineare nel numero delle foglie riportate (perché il numero di nodi interni di un albero binario è minore del numero delle foglie).

- In totale quindi si deve aggiungere al costo della ricerca un termine pari al numero di punti che cadono in $\mathcal{R}$.

- Riassumendo: in un k-d tree bilanciato il Range Search - Count costa $O(\sqrt{n})$, mentre la Range Search - Report costa $O(\ell + \sqrt{n})$, dove ℓ è il numero di punti che cadono nella regione di ricerca.

- ***Discussione***: i k-d tree occupano spazio lineare (ottimale), ma hanno un costo di interrogazione elevato. Sono comunque molto usati in pratica.

- Possiamo congetturare che si possa barattare una maggior occupazione spaziale con un tempo di interrogazione migliore, ed infatti esistono i ***Range Trees*** che occupano $O(n \log n)$ spazio, ma con un tempo di interrogazione di $O(\log^2 n + \ell)$.

- Una generalizzazione dei k-d tree rimuove il vincolo che il partizionamento avvenga lungo rette ortogonali. Nei ***binary space partition (BSP)*** tree, le linee di suddivisione sono rette (iperpiani) generici e le celle sono poligoni convessi. I BSP tree però non sono utili a risolvere problemi di ricerca ortogonale, ovviamente.

Prossimità

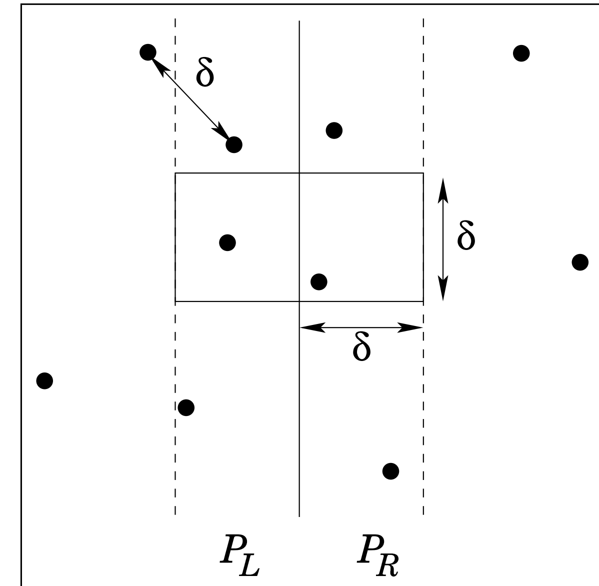
- Si tratta di problemi che coinvolgono il concetto di prossimità o vicinanza tra punti.
- ***Problema 1:*** (Closest Pair) Dato un insieme $\mathcal{S}$ di n punti, trovare i due più vicini.
- ***Problema 2:*** (All Nearest Neighbours) Dato un insieme $\mathcal{S}$ di n punti nel piano, trovare il più vicino di ciascuno.
- Quest'ultimo problema si generalizza al seguente:
- ***Problema 3:*** (Loci of Proximity) Dato un insieme $\mathcal{S}$ di n punti nel piano, per ciascun punto P determinare il luogo dei punti del piano che sono più vicini a P che a qualunque altro punto di $\mathcal{S} \setminus \{P\}$.
- Vedremo un algoritmo *divide-et-impera* per risolvere Closest Pair e studieremo una struttura dati notevole che deriva dal Loci of Proximity: i ***diagrammi di Voronoi***.
- Questi ultimi ci condurranno allo studio della ***triangolazione di Delaunay***,

Algoritmo di Shamos

- ***Risolve*** 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 ordinando i punti di $\mathcal{S}$ separatamente in x e in y (due vettori).
- L'algoritmo ricorsivo segue lo schema classico del divide-et-impera. Si articola quindi in tre fasi:

Dividi: biseca con una linea retta verticale l'insieme di punti $\mathcal{S}$ in due sottoinsiemi $\mathcal{S}_L$ e $\mathcal{S}_R$, in modo che i punti di $\mathcal{S}_L$ stiano alla sinistra della retta e quelli di $\mathcal{S}_R$ stiano alla destra della retta. Nella pratica si devono suddividere i due vettori in x e y (in tempo lineare).

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

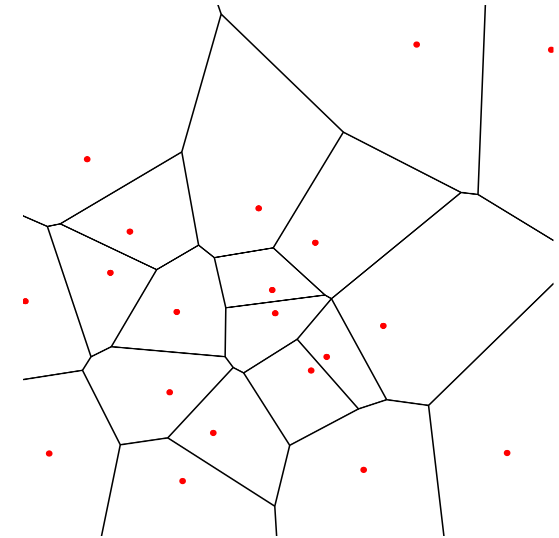


Combina: la coppia di punti più vicini in $\mathcal{S}$ è una delle due coppie trovate in $\mathcal{S}_L$ e $\mathcal{S}_R$ oppure una coppia formata da un punto di $\mathcal{S}_L$ ed uno di $\mathcal{S}_R$. Detta δ la minima distanza tra le coppie ritornate dalla chiamata ricorsiva, bisogna determinare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

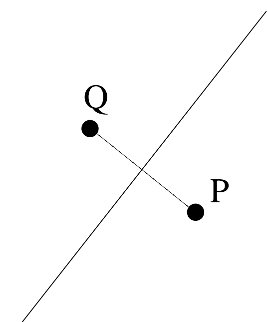
L'osservazione chiave è che tali punti sono in numero ***costante***, infatti, in un rettangolo $2\delta \times \delta$ possono esserci al più 6 punti separati almeno da una distanza δ. Per ogni punto della striscia, solo i 5 punti che lo seguono nel vettore ordinato per y crescente devono essere considerati.

- ***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



- La soluzione al LOCI OF PROXIMITY è una partizione del piano in regioni. Ciascuna regione, detta cella di Voronoi $\mathcal{V}(P)$, è definita come l'insieme dei punti del piano Q tali che $\forall R \in \mathcal{S} \setminus \{P\}$ si abbia $\|Q - P\| < \|Q - R\|$.
- Dati due punti P e Q, $\mathcal{V}(P)$ è il semipiano aperto contenente P definito dalla bisettrice di $\overline{PQ}$.
- La bisettrice di un segmento $\overline{PQ}$ è la retta perpendicolare a $\overline{PQ}$ nel punto medio.
- In generale, la cella di Voronoi $\mathcal{V}(P)$ si ottiene come intersezione dei semipiani definiti dalle bisettrici dei segmenti che uniscono P con tutti gli altri punti.

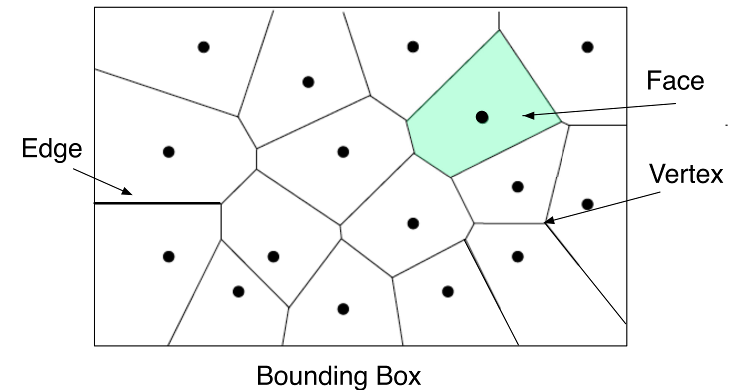


- Dunque la cella di Voronoi è una regione poligonale convessa.

- ***Proposizione.*** Le celle di Voronoi illimitate sono tutte e sole quelle dei punti appartenenti alla frontiera del guscio convesso di $\mathcal{S}$. (dim. su [1])

- La partizione del piano in un numero finito di regioni poligonali convesse indotto dalle celle di Voronoi dell'insieme di punti $\mathcal{S}$ prende il nome di ***diagramma di Voronoi*** (DV).

- Nota: Il DV non è, a rigore, una suddivisione piana, ma racchiudendo $\mathcal{S}$ in un rettangolo che contenga tutti i punti (e quindi anche il loro guscio convesso), possiamo rendere limitate le regioni illimitate, ottenendo quindi un PSLG, ovvero una suddivisione piana.



- Questo permette di rappresentare un DV con un DCEL, per esempio.

- Parleremo quindi di vertici, spigoli e facce (ovvero celle) del diagramma di Voronoi. Come per le suddivisioni piane, facce e spigoli sono aperti.

- Ipotesi di ***posizione generale***: in $\mathcal{S}$ non vi sono 4 punti conciclici (ovvero che giacciono sulla medesima circonferenza).

- ***Osservazione***: ogni spigolo del diagramma di Voronoi è un segmento di bisettrice di una coppia di punti di $\mathcal{S}$, e dunque vi incidono esattamente due facce.

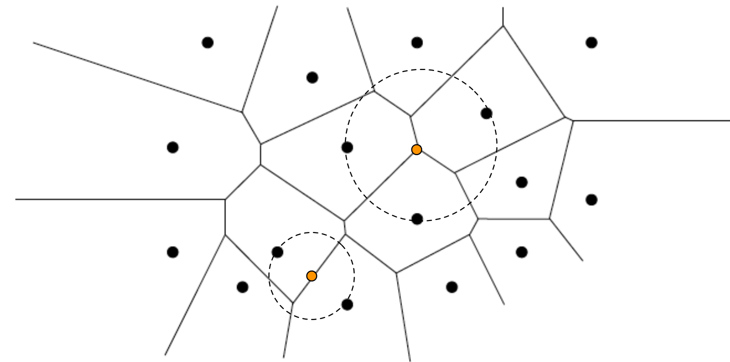
- ***Lemma 1***: su ciascun vertice del diagramma di Voronoi incidono tre spigoli (e dunque tre facce). (dim. su [1])

- Un vertice Q 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 una circonferenza centrata in Q che passa per P_1, P_2 e P_3, chiamiamola $\mathcal{C}$.

- ***Lemma 2***: $\mathcal{C}$ non contiene nessun punto di $\mathcal{S}$ al suo interno.

- Dim. Se P_4 fosse interno a $\mathcal{C}$, allora P_4 sarebbe più vicino a Q di ciascuno tra P_1, P_2 e P_3, quindi Q dovrebbe appartenere a $\mathcal{V}(P_4)$ e non a $\mathcal{V}(P_1)$, $\mathcal{V}(P_2)$ e $\mathcal{V}(P_3)$.

- ***Definizione*** Una circonferenza con questa proprietà (ovvero che non contiene punti di $\mathcal{S}$ al suo interno) si dice ***$\mathcal{S}$-libera***.



- ***Teorema***. (i) La bisettrice tra due punti A e B di $\mathcal{S}$ definisce uno spigolo del DV di $\mathcal{S}$ se solo se esiste una circonferenza $\mathcal{S}$-libera centrata su un punto della bisettrice e che passa per A e B. (ii) Un punto P è vertice del DV di $\mathcal{S}$ se e solo se esiste una circonferenza $\mathcal{S}$-libera centrata in P passante per tre punti di $\mathcal{S}$.

- Dim. (i) Se A e B sono adiacenti in DV, sia Q un punto dello spigolo che separa $\mathcal{V}(A)$ e $\mathcal{V}(B)$. Allora la circonferenza centrata in Q e passante per A e B deve essere $\mathcal{S}$-libera (argomento come Lemma 2). Viceversa, se la circonferenza è libera, si ha che $\text{dist}(Q, A) = \text{dist}(Q, B) \leq \text{dist}(Q, P) \quad \forall P \in \mathcal{S}$, quindi Q giace sullo spigolo che divide $\mathcal{V}(A)$ e $\mathcal{V}(B)$. (ii) Un verso è il Lemma 2. L'altro verso: il centro P della circonferenza è equidistante dai tre punti, quindi giace simultaneamente sulle tre bisettrici. Le bisettrici sono spigoli grazie al fatto che la circonferenza è $\mathcal{S}$-libera, per quanto dimostrato in (i).

- Si può ***calcolare il diagramma di Voronoi*** usando la definizione in termini di intersezione di semispazi.

- La cella $\mathcal{V}(P_i)$ viene calcolata come intersezione degli $n-1$ semipiani che hanno come retta di supporto la bisettrice di $\overline{P_i P_j} \quad \forall P_j \in \mathcal{S} \setminus \{P_i\}$. Il costo totale è $O(n^2 \log n)$.

- Vedremo che si può fare di meglio. In particolare verranno illustrati algoritmi per il calcolo del DV in $O(n \log n)$. Siccome dal DV si estrarre il guscio convesso, questa complessità è ***ottimale***.

- Accenniamo all'algoritmo "divide-et-impera" e poi vediamo in maggior dettaglio l'algoritmo di Fortune (*plane sweep*).

Algoritmo divide-et-impera

(cenni)

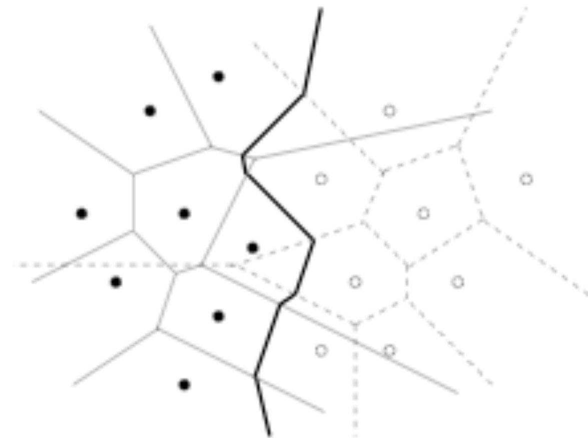
- ***Risolve*** Loci of Proximity (ovvero calcola il DV).
- L'algoritmo ricorsivo segue lo schema classico del divide-et-impera. Si articola quindi in tre fasi:

  Dividi: Partiziona $\mathcal{S}$ con una retta verticale lungo la mediana delle ascisse in due insiemi $\mathcal{S}_L$ ed $\mathcal{S}_R$.

  Domina: Calcola ricorsivamente i DV di $\mathcal{L}$ ed $\mathcal{R}$.

  Combina: Unisci i due diagrammi per ottenere DV

- Il lavoro si svolge nella fase "combina".
  Consiste nel calcolare la catena poligonale (monotona in y) che separa i due DV di $\mathcal{S}_L$ ed $\mathcal{S}_R$.



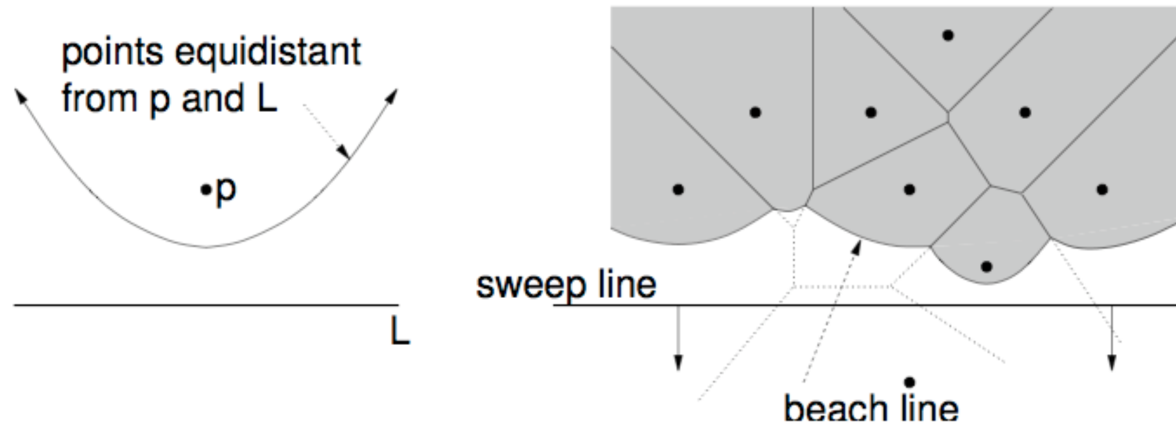
- ***Complessità***: assumendo di poter eseguire la fase "combina" in tempo lineare nel numero dei vertici, il costo computazionale dell'algoritmo è dato dalla (solita) ricorrenza: $T(n) = 2T(n/2) + O(n)$, la cui soluzione è $O(n \log n)$.

L'algoritmo di Fortune

- ***Risolve*** LOCI OF PROXIMITY (ovvero calcola il DV).
- Sia dato l'insieme $\mathcal{S}$ di n punti sul piano di cui si vuol trovare il diagramma di Voronoi. Li chiameremo ***siti***.
- Sia ℓ una retta orizzontale che spazzola il piano dall'alto verso il basso denominata ***sweep line*** (per l'intersezione di segmenti avevamo usato una retta verticale). Sia ℓ^+ il semipiano chiuso superiore a ℓ.
- Nell'algoritmo di Bentley-Ottmann, quando la *sweep line* incontra un evento questo non modifica la parte di piano già visitata dalla *sweep line*.
- Nel caso del diagramma di Voronoi è facile vedere che ci sono vertici del diagramma di Voronoi (che sono gli oggetti che cerchiamo) che cadono in ℓ^+ e tuttavia sono influenzati da punti non ancora raggiunti da ℓ.
- L'algoritmo di Fortune aggira questo problema introducendo un'ulteriore struttura denominata (si capirà perché) ***beach line***.

- La *beach line* si muove assieme alla *sweep line*, rimanendo dietro di quest'ultima. La porzione del DV che giace al di sopra della *beach line* è definitivo, nel senso che non viene influenzato dai siti al di sotto della *sweep line* (ancora da scoprire).

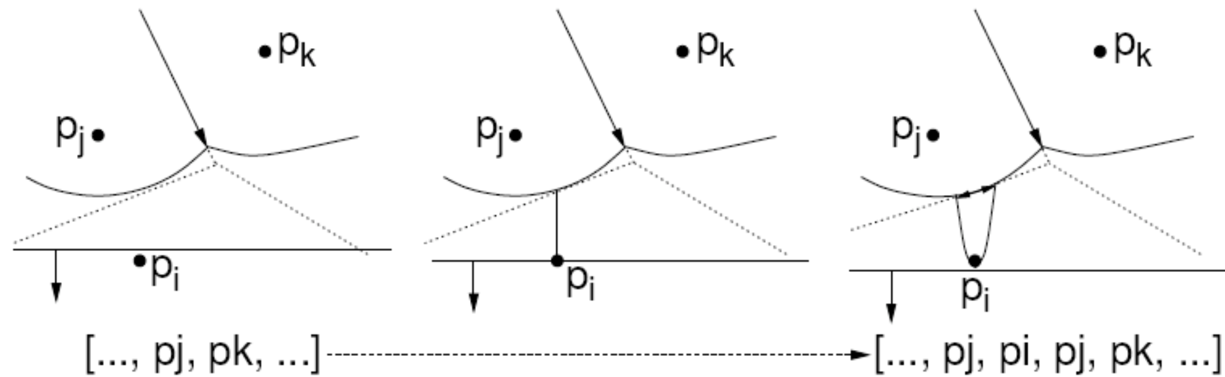
- La ***beach line*** è definita come il confine tra due regioni di ℓ^+: quella formata dai punti del piano più vicini ad un qualche sito di ℓ^+ che a ℓ stessa e quelli più vicini al ℓ che ai siti di ℓ^+.



- Dunque la ***beach line*** è il luogo dei punti equidistanti dalla *sweep line* ℓ e dal sito più vicino di ℓ^+.

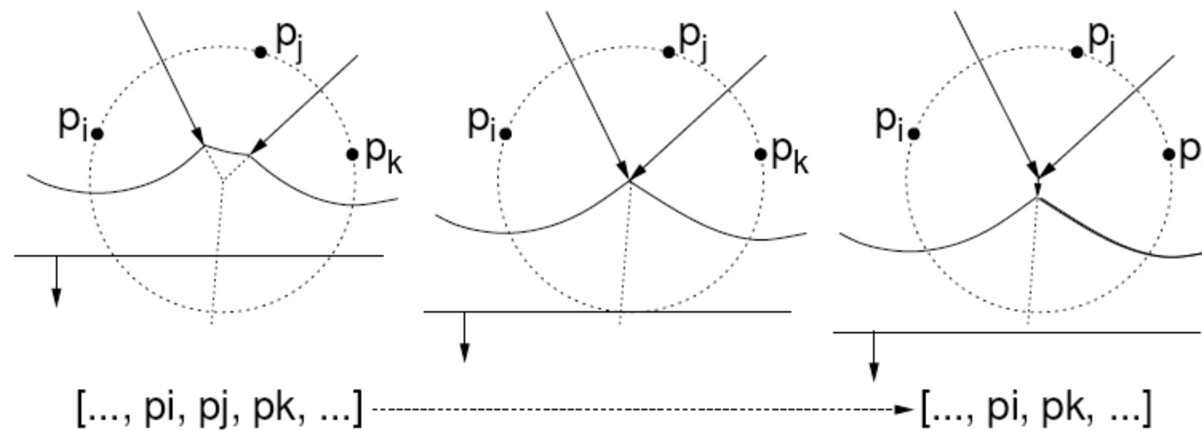
- Il luogo dei punti equidistanti da un punto e una retta orizzontale è una parabola con asse verticale. Quindi la *beach line* è una curva x-monotona e che è formata da archi di parabola.
- Tali archi di parabola sono associati ciascuno ad un qualche sito di ℓ^+.
- I ***punti singolari*** della *beach line*, ovvero i punti dove due archi di parabola si incontrano, sono equidistanti da due siti di ℓ^+ (e da ℓ), quindi giacciono su uno spigolo del DV di $\mathcal{S}$.
- Se immaginiamo di muovere ℓ con continuità verso il basso, il percorso tracciato dai punti singolari della *beach line* sono gli spigoli del DV (v. applet).
- In realtà, si considera solo lo spostamento di ℓ in un insieme discreto posizioni, corrispondenti agli ***eventi*** nei quali la *sweep line* cambia il suo stato.
- Lo ***stato*** della *sweep line* comprende la sua ordinata corrente e la lista ordinata da sinistra a destra dei siti che definiscono la *beach line*.

- Gli eventi sono di due tipi:



  – ***Evento di sito:*** quando ℓ attraversa un nuovo sito di $\mathcal{S}$, un nuovo arco parabolico viene aggiunto alla *beach line*. Questi eventi sono tutti noti a priori, basta ordinare per y decrescenti gli elementi di $\mathcal{S}$.

  – L'aggiunta del nuovo arco al più ne spezza uno esistente, con un incremento netto di due archi per sito. Al massimo quindi ci sono $2n - 1$ archi nella *beach line*.



– ***Evento di vertice:*** Quando due punti singolari della *beach line* si incontrano, allora il punto di incontro è un vertice del diagramma di Voronoi (perché il punto è equidistante da tre siti di ℓ^+ e da ℓ). La proprietà chiave è che questi eventi sono generati da archi che sono vicini nella *beach line*. In particolare, solo triple di archi consecutivi nella *beach line* possono generare un evento di vertice. Quando la circonferenza passante per i tre siti associati agli archi è tangente alla *sweep line*, l'arco centrale centrale sparisce ed i due archi estremi si intersecano in un vertice del DV.

– Generazione di un evento di vertice: per ogni tripletta di siti consecutivi nella *beach line* si calcola il cerchio passante per tali punti; se il punto inferiore Q di tale cerchio giace sotto ℓ allora si crea un evento di vertice di ordinata Q_y.

- Anche le strutture dati necessarie sono due:
  - ***La beach line:*** è concettualmente una lista di archi parabolici. In realtà non c'è bisogno di descrivere gli archi parabolici, basta mantenere una lista dei siti di $\mathcal{S}$ corrispondenti agli archi. In genere si usa un albero binario bilanciato. Le operazioni che la *beach line* deve permettere sono
    1. Determinare l'arco della *beach line* che si trova sopra un determinato punto della *sweep line* (serve per gli eventi di sito)
    2. Calcolare il precedente ed il successivo lungo la *beach line*
    3. Inserire un nuovo arco
    4. Cancellare un arco dalla *beach line*
  - ***La coda degli eventi:*** è la coda degli eventi (di sito e di vertice) futuri; deve poter permettere l'inserzione di nuovi eventi, l'eliminazione di eventi e l'estrazione del primo evento (y più alta). Ad ogni evento va associata anche la sua y. Ognuno di questi eventi di vertice possiede dei puntatori alla tripletta di siti della *beach line* che lo hanno generato

- Si noti che gli eventi di vertice presenti nella coda in un certo istante sono solo "potenziali", ovvero la tripla che li ha generati potrebbe scomparire dalla *beach line* prima che ℓ raggiunga l'evento stesso. In tal caso l'evento di vertice viene rimosso dalla coda degli eventi.

- L'algoritmo di Fortune procede quindi nel seguente modo

  1. Si estrae un evento dalla coda degli eventi
     - se è un ***evento di sito***, sia P_i il sito incontrato da ℓ e sia P_j il sito corrispondente all'arco della *beach line* che deve essere diviso. Si rimpiazza P_j con la tripla P_j, P_i, P_j nella *beach line*. Nella coda degli eventi si eliminano gli eventi di vertice nati da triplette che avevano P_j al centro e si accodano nuovi eventi di vertice per le nuove triplette che si sono formate.
     - se è un ***evento di vertice*** si inserisce il nuovo vertice, si considerano i tre siti P_i, P_j, P_k che lo hanno generato e si elimina P_j dalla *beach line*. Nella coda degli eventi si eliminano tutti gli eventi di vertice nati da triplette che contengono P_j e si accodano nuovi eventi di vertice per le nuove triplette contenenti P_j e P_k.

- ***Complessità***: gli eventi sono $O(n)$. Le strutture dati occupano $O(n)$ ad ogni passo, quindi gli accessi per aggiornarle costano $O(\log n)$. In totale il tempo speso dell'algoritmo di Fortune è $O(n \log n)$.

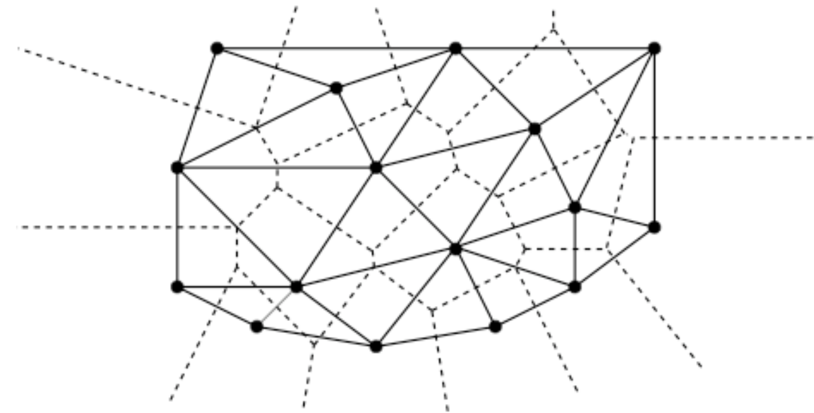
- Grazie alla costruzione del DV, si possono risolvere i seguenti problemi:
- ALL NEAREST NEIGHBOURS: il punto più vicino a P si trova esaminando le celle adiacenti a $V(P)$ (una per ogni spigolo della cella). Poiché ogni spigolo viene visitato al più due volte (ogni spigolo incide su al più due celle) e gli spigoli sono $O(n)$, il costo è lineare.
- ***Problema:*** (NEAREST NEIGHBOUR SEARCH) Dato un insieme $\mathcal{S}$ di n punti nel piano, e dato un nuovo punto Q (di interrogazione) si chiede di trovare il punto dell'insieme $\mathcal{S}$ che è più vicino a Q.
- Basta cercare la cella in cui cade Q (problema di PLANAR POINT LOCATION).
- Nota. NEAREST NEIGHBOUR SEARCH si può risolvere anche costruendo un k-d tree. Nel caso peggiore richiede $O(n^2)$ tempo, ma nel caso medio è $O(2^d + \log n)$.
- Si effettua una ricerca sull'albero e mantenendo il punto più vicino corrente. Un sottoalbero viene scartato quando si giudica che tutti i punti in esso contenuti distano dal punto di interrogazione più del risultato corrente.

Applicazioni dei DV

- ***Nearest neighbour clustering*** (pattern recognition). Alcuni oggetti campione sono punti in un opportuno *feature space*. Un nuovo oggetto da classificare viene assegnato al campione più vicino. Le classi definite dai campioni in questo modo sono celle di Voronoi.
- ***Facility location***. Un nuovo negozio dovrà essere posizionato il più lontano possibile da quelli esistenti. Ovvero, al centro di una circonferenza libera di raggio massimo. I vertici del DV sono i potenziali siti.
- ***Path planning*** (robotica). Un robot che vuole rimanere il più distante possibile dagli ostacoli dovrà muoversi sul DV di questi ultimi.
- ***Cristallografia***. Supponiamo che un certo numero di semi di cristalli (nel piano) crescano ad una velocità uguale e costante. Ogni seme cresce fino a quando trova spazio libero (non occupato già da un altro cristallo). Quindi ogni seme cresce fino a riempire la propria cella di Voronoi.

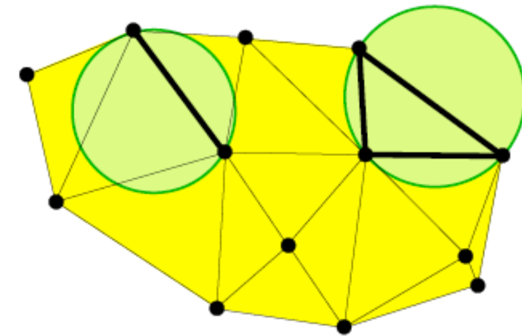
Triangolazione di Delaunay

- Sia $\mathcal{S}$ un insieme di punti nel piano.
- Consideriamo il PSLG ***duale*** del diagramma di Voronoi di $\mathcal{S}$, ovvero il grafo a spigoli rettilinei ottenuto congiungendo tutti i punti di $\mathcal{S}$ le cui celle di Voronoi sono adiacenti.



- ***Teorema*** (Delaunay): Il PSLG duale del diagramma di Voronoi è una triangolazione di $\mathcal{S}$.
- Dim. Poiché i vertici del diagramma di Voronoi hanno grado 3 (grazie all'assunzione di posizione generale), le facce (limitate) del grafo duale sono triangoli. (v. [1]).

- Il duale del diagramma di Voronoi prende il nome di grafo o (in virtù del Teorema) ***triangolazione di Delaunay*** (TD).

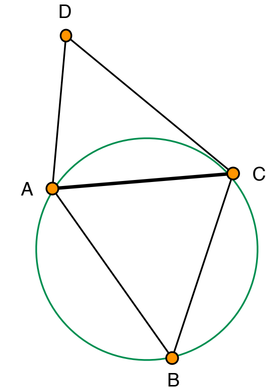


- ***Lemma 1***. Esiste una circonferenza $\mathcal{S}$-libera che passa per due vertici di $\mathcal{S}$ se e solo se i due vertici incidono sullo stesso spigolo della TD.
- ***Lemma 2***. La circonferenza che passa per tre vertici di $\mathcal{S}$ è $\mathcal{S}$-libera se e solo se i tre vertici incidono sulla stessa faccia della TD.
- Dim. I due Lemmi derivano immediatamente dalla dualità con il diagramma di Voronoi.

- ***Teorema***: (caratterizzazione tramite il ***criterio della circonferenza***) una triangolazione di un insieme di punti $\mathcal{S}$ è di Delaunay se e solo se (i) ogni triangolo è inscritto in una circonferenza $\mathcal{S}$-libera se e solo se (ii) ciascuno spigolo ammette una circonferenza $\mathcal{S}$-libera che passa per i suoi estremi.
- Dim. (Del. $\Rightarrow$ (i) $\Rightarrow$ (ii) $\Rightarrow$ Del). Se la triangolazione è di Delaunay, allora ogni triangolo è inscritto in una circonferenza $\mathcal{S}$-libera, per il Lemma 2, e dunque anche tutti gli spigoli hanno una circonferenza $\mathcal{S}$-libera. L'ultima implicazione: $\neg$Del $\Rightarrow$ $\neg$(ii). Se la triangolazione non è di Delaunay esiste almeno uno spigolo che non appartiene a TD. Per il Lemma 1 tale spigolo non ammette una circonferenza $\mathcal{S}$-libera.

- ***Definizione***: In virtù del Teorema, un triangolo inscritto in una circonferenza $\mathcal{S}$-libera si dice ***di Delaunay***, così come uno spigolo $\overline{AB}$ che ammette una circonferenza $\mathcal{S}$-libera passante per A e B si dice ***di Delaunay***,

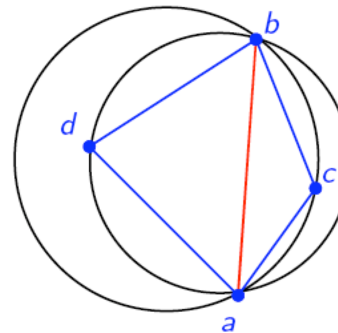
- ***Definizione***: Siamo ABC e ACD due facce interne adiacenti in una triangolazione. I vertici $ABCD$ individuano un poligono semplice che ha $\overline{AC}$ come diagonale. Lo spigolo $\overline{AC}$ è ***localmente Delaunay*** se la circonferenza passante per ABC non contiene D.



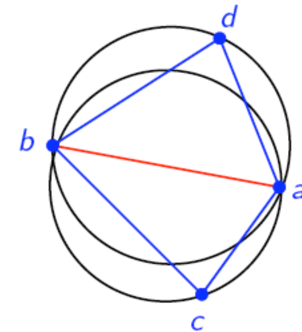
- ***Oss.*** Al posto di D avremmo potuto prendere B. Basta controllarne uno solo poiché:

- Se $ABCD$ è un quadrilatero convesso, si ha che: B è esterno alla circonferenza CDA e D è esterno alla circonferenza ABC oppure B è interno alla circonferenza CDA e D è interno alla circonferenza ABC.

abcd convex quadrilateral.



or



- Se Se $ABCD$ non è convesso, sia AC la (unica) diagonale. Allora B è esterno alla circonferenza ACD e D è esterno alla circonferenza ABC.

  - Si noti che poiché i triangoli fanno parte di una triangolazione più grande, uno spigolo localmente Delaunay potrebbe non essere di Delaunay, per colpa di qualche altro punto di $\mathcal{S}$ diverso da B o D. Tuttavia vale il seguente:

- ***Lemma 3*** Se in una triangolazione tutti gli spigoli sono localmente Delaunay, allora tutti gli spigoli sono di Delaunay. (dunque la triangolazione è di Delaunay). (dim. su [7])

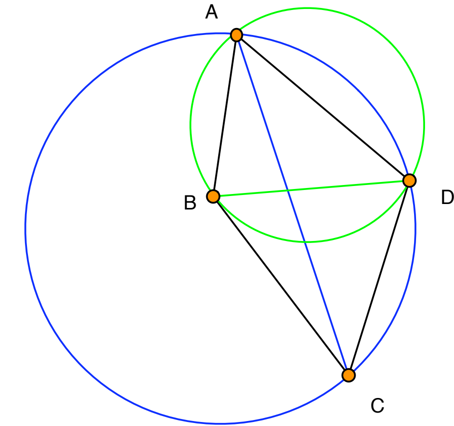
Flip Algorithm

1 Si parta con una triangolazione di $\mathcal{S}$ qualunque se ne esaminino gli spigoli.

2 Ogni volta che se ne trova uno non localmente Delaunay lo si ***ribalti*** (*flip*) facendolo divenire localmente Delaunay.

3 Si termina quando tutti gli spigoli sono localmente Delaunay (e dunque la triangolazione è di Delaunay).

- L'operazione di ***ribaltamento*** è resa possibile dal seguente:



- ***Lemma 4*** (edge flipping): Sia $\overline{AC}$ uno spigolo di una triangolazione, e siano ABC e ACD le due facce interne ad esso adiacenti. Allora: $\overline{AC}$ è localmente Delaunay oppure $\overline{AC}$ è "ribaltabile", ovvero: il quadrilatero $ABCD$ generato dalla rimozione di $\overline{AC}$ è convesso e la diagonale $\overline{BD}$ è localmente Delaunay (dim. su [7]).

- Dunque il ribaltamento consiste nel sostituire $\overline{AC}$ con $\overline{BD}$.

- ***Complessità*** Le coppie triangolo-punto che possono violare il criterio della circonferenza sono $O(n^2)$. Ogni ribaltamento riduce di almeno due tale numero (v. [7]), quindi il processo termina e servono $O(n^2)$ ribaltamenti.

- ***Operazione di base***. Come si controlla la proprietà di "localmente Delaunay", ovvero come si verifica se una circonferenza è $\mathcal{S}$-libera?

- Formalizziamo il concetto di circonferenza libera definendo un predicato booleano che si applica a quattro punti distinti del piano: $\mathrm{InCircle}(A, B, C, D)$ è vero se e solo se il punto D è interno alla circonferenza ABC.

- Siano A, B, C e D quattro punti sul piano; si dimostra (più avanti) che $\mathrm{InCircle}(A, B, C, D)$ è vero se e solo se

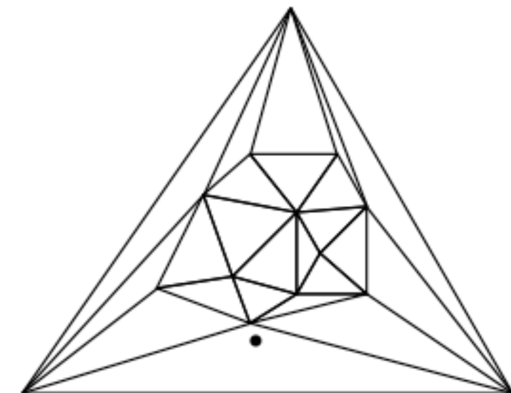
$$\det \begin{pmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{pmatrix} > 0$$

Proprietà del max-min angolo

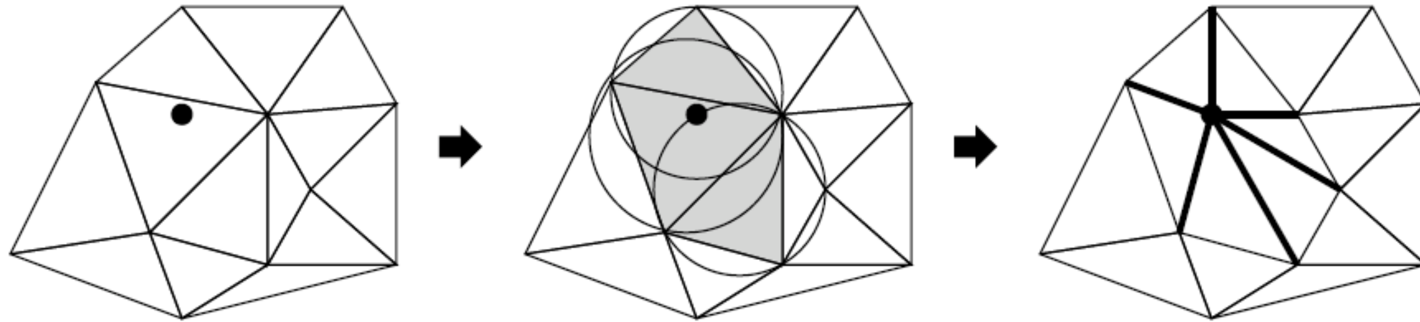
- ***"Localmente Delaunay equivale a max-min angolo"***. Con riferimento al Lemma 4 (edge flipping), consideriamo il quadrilatero convesso $ABCD$. Si dimostra che scegliere la diagonale localmente Delaunay tra le due possibili equivale a massimizzare il più piccolo angolo interno dei due triangoli risultanti (dim. su [5], segue dal teorema di Talete).
- Possiamo pensare di ordinare le triangolazioni di $\mathcal{S}$ in base all'angolo più piccolo, ed a parità di questo guardare il secondo più piccolo, e così via in un ordinamento lessicografico.
- Dunque l'operazione di ribaltamento degli spigoli incrementa il rango della triangolazione nell'ordinamento, perché aumenta il suo angolo più piccolo.
- Poiché le triangolazioni sono in numero finito, il processo di ribaltamento degli spigoli termina (lo sapevamo già), e la triangolazione risultante (che è di Delaunay) ha la proprietà di possedere, tra tutte le triangolazioni, il ***massimo angolo minimo***.
- Quindi la TD ha la proprietà di avere gli angoli meno acuti possibile.

Algoritmo incrementale

- ***Risolve***: calcolo della TD.
- Naturalmente la TD si può calcolare dal DV e viceversa. Per il DV abbiamo visto un approccio "divide-et-impera" ed uno *plane sweep*. Vediamo quindi un approccio incrementale per la TD.
- ***Idea generale***
  - I vertici di $\mathcal{S}$ vengono inseriti uno alla volta. Dopo ogni inserimento la triangolazione parziale viene "aggiustata" in modo da continuare ad essere di Delaunay.
  - Si parte con un super triangolo fittizio che contiene tutti i punti. Alla fine i vertici del super triangolo e gli spigoli incidenti vengono rimossi.
  - Ha stessa complessità del *flip algorithm* nel caso peggiore ($O(n^2)$), ma si riesce a dimostrare un costo ***atteso*** di $O(n \log n)$ nella versione randomizzata.

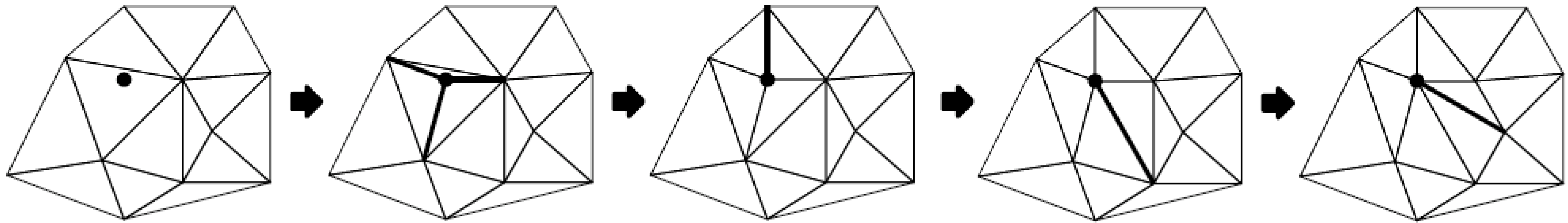


- ***Versione di Bowyer-Watson***

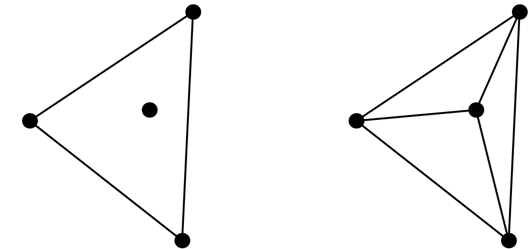


  - Quando viene inserito un vertice P si rimuovono tutti i triangoli nella cui circonferenza circoscritta cade P.
  - La cavità poligonale che si produce viene triangolata collegando P a ciascun vertice della cavità.
  - I nuovi spigoli sono Delaunay per costruzione (v. [7]).
  - Si noti che tutti i nuovi spigoli creati in seguito all'inserimento di P sono incidenti in P.

- ***Versione di Lawson***



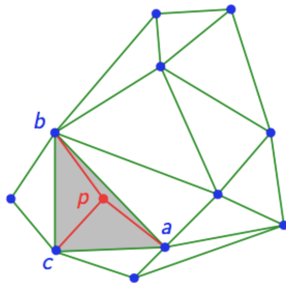
  - L'algoritmo di Lawson produce lo stesso risultato di quello di Bowyer-Watson, ma si basa sul ribaltamento (*flip*) degli spigoli e non richiede di individuare la cavità.
  - Sia ABC il triangolo in cui cade il vertice P appena inserito.
  - Il triangolo ABC viene suddiviso in 3 nuovi triangoli con l'aggiunta degli spigoli PA, PB e PC.



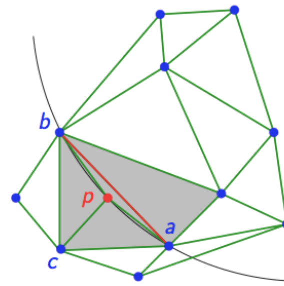
  - I tre nuovi spigoli sono localmente Delaunay per costruzione, poiché i triangoli incidenti formano un quadrilatero non convesso. Bisogna quindi controllare gli spigoli AB, BC e CD (quelli ***opposti*** a P).
  - Se qualcuno di essi non è localmente Delaunay, allora viene ribaltato, e gli spigoli ***opposti*** a P dei due nuovi triangoli vengono controllati ricorsivamente.

- Basta controllare solo gli spigoli opposti a P perché solo questi, se ribaltati, possono dare uno spigolo che ha P per estremo.
- Pseudocodice:

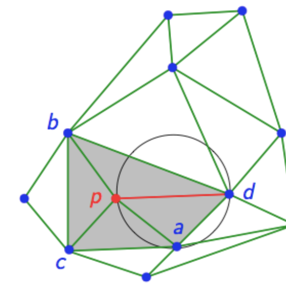
```
Insert(P) {
     Find the triangle ABC containing P;
     Insert edges PA, PB, and PC into triangulation;
     FixEdge(AB); // Fix the surrounding edges
     FixEdge(BC);
     FixEdge(CA);
}
FixEdge(P,AB) {
// Test AB and flip it if necessary
    if (AB is an edge on the exterior face) return;
    Let ADB the triangle adjacent ABP along  AB;
    if (inCircle(P,A,B,D) { // D violates the incircle test
         Flip edge AB for PD;
         FixEdge(P,AD); // Fix the new suspect edges
         FixEdge(P,DB);
    }
}
```



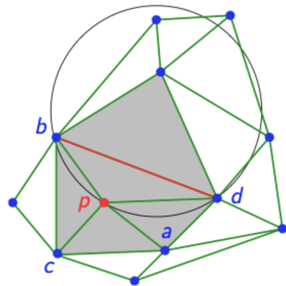
Inserito p



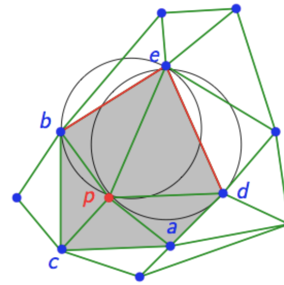
ab non è loc. Del.



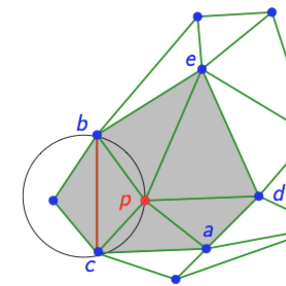
flip ab, ad è loc. Del.



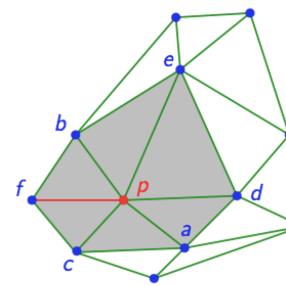
bd non è loc. Del.



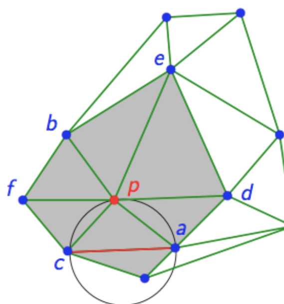
flip bd, de e be sono loc. Del.



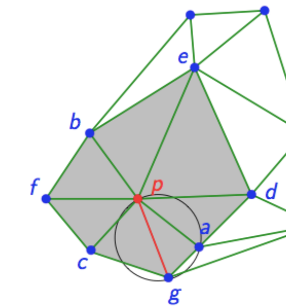
bc non è loc. Del.



flip bc



ac non è loc. Del.

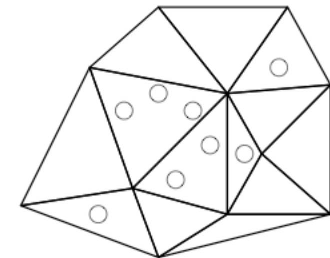


fip ac, ag è loc. Del. Fine

- ***Importante***: L'inserimento di un punto richiede la localizzazione del triangolo a cui appartiene (in effetti questa operazione domina la complessità dell'algoritmo).

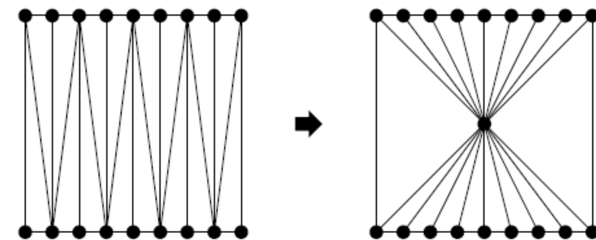
- Bisogna mantenere una opportuna struttura dati. Quella che vediamo deriva dalla semplificazione del ***conflict graph*** (v. p.es. [7])

- Ogni triangolo della triangolazione corrente è un ***bucket*** (secchio, cesto) che contiene i punti non ancora inseriti che cadono in quel triangolo.



- Ogni volta che un triangolo viene suddiviso per l'inserimento di un punto o che uno spigolo viene ribaltato alcuni triangoli vecchi vengono eliminati ed alcuni nuovi vengono creati. I punti vengono riassegnati di conseguenza.

- ***Complessità***. Il numero di ribaltamenti necessari per accomodare un nuovo punto può arrivare a $O(k)$ dove k è il numero di vertici presenti nella triangolazione. Il costo per riassegnare un punto è costante, ed al massimo un punto può venire riassegnato ad ogni inserimento, quindi $O(n)$ volte. Quindi nel caso peggiore il costo è $O(n^2)$.



- Nella pratica, escludendo i casi patologici, si osserva una complessità minore, dovuta ad una sorta di località delle operazioni richieste per l'inserimento di un punto.

- Questa osservazione si può rendere più precisa considerando la versione ***randomizzata*** dell'algoritmo di Lawson, in cui i punti vengono inseriti a caso.

- L'analisi che si effettua segue la modalità della ***backward analysis***. Si consideri il lavoro necessario per l'inserimento dell'ultimo punto P. Poiché ogni ribaltamento aggiunge uno spigolo incidente in P, il numero di ribaltamenti necessari per inserire P è pari al suo grado meno 3 (ci sono 3 spigoli inizialmente).

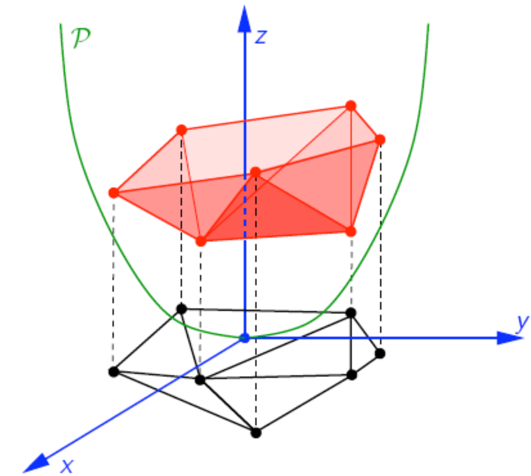
- Dato che il grado medio di un grafo piano è 6, l'inserimento di P (esclusa la localizzazione) ha un costo atteso costante ($O(1)$). Grazie alla randomizzazione, ciascun punto ha la stessa probabilità di essere l'ultimo, quindi l'inserimento di un qualunque punto ha un costo atteso costante.

- Per quanto riguarda la localizzazione, su [6] si dimostra che un punto viene riassegnato $O(\log n)$ volte (valore atteso) e dunque il costo totale per la localizzazione è $O(n \log n)$. Dunque, complessivamente il costo ***atteso*** è $O(n \log n)$.

- Cos'è il costo atteso? Vuol dire che, con elevata probabilità l'algoritmo ha questo costo su ogni input.

Sollevamento e guscio convesso

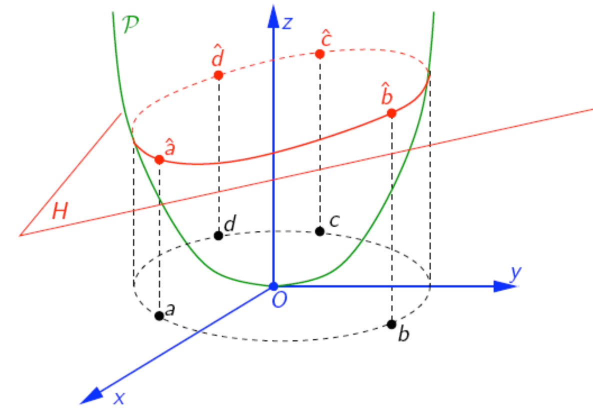
- Esiste una (sorprendente) connessione tra la TD in E^2 ed il guscio convesso in E^3, tramite il paraboloide 3D $z = x^2 + y^2$.
- Per ogni punto nel piano $P = (P_x, P_y)$, la sua proiezione verticale sul paraboloide (***sollevamento***) è il punto 3D $\tilde{P} = (P_x, P_y, P_x^2 + P_y^2)$.
- Dato un insieme $\mathcal{S}$ di punti nel piano, sia $\tilde{\mathcal{S}}$ il sollevamento di tutti i punti di $\mathcal{S}$.
- ***Osservazione***: Le sezioni con piani non verticali del paraboloide sono ellissi che si proiettano su cerchi nel piano (x, y) (si verifica analiticamente).
- ***Lemma*** Siano A, B, C, D quattro punti del piano, e siano $\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D}$ le loro rispettive proiezioni sul paraboloide $z = x^2 + y^2$. Il punto D giace all'interno della circonferenza per A, B, C se e solo se $\tilde{D}$ giace sotto al piano passante per $\tilde{A}, \tilde{B}, \tilde{C}$.



- Per verificare se un punto è interno alla circonferenza, basta verificare se la quadrupla $\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D}$ è orientata in senso antiorario. Quindi $\text{InCircle}(P, Q, R, S) \Leftrightarrow \text{orient}(\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D}) > 0$, da cui segue la formula usata per InCircle.

- *Definizione* Il guscio convesso inferiore di $\tilde{\mathcal{S}}$, che denotiamo con $\mathrm{LCH}(\tilde{\mathcal{S}})$, è la porzione del guscio che si vede da $z = -\infty$. Si tratta di un poliedro convesso.

- *Teorema*. Dato un insieme di punti del piano $\mathcal{S}$ in posizione generale. Siano $\mathrm{A}, \mathrm{B}, \mathrm{C} \in \mathcal{S}$ e siano $\tilde{\mathrm{A}}, \tilde{\mathrm{B}}, \tilde{\mathrm{C}}$ le rispettive proiezioni di questi punti sul paraboloide. Allora il triangolo ABC è una faccia della TD di $\mathcal{S}$ se e solo se il triangolo $\tilde{\mathrm{A}}\tilde{\mathrm{B}}\tilde{\mathrm{C}}$ è una faccia di $\mathrm{LCH}(\tilde{\mathcal{S}})$.



- Dim. Tre punti $\tilde{\mathrm{A}}, \tilde{\mathrm{B}}, \tilde{\mathrm{C}} \in \tilde{\mathcal{S}}$ formano una faccia di $\mathrm{LCH}(\tilde{\mathcal{S}})$ se e solo se non vi sono punti di $\tilde{\mathcal{S}}$ al di sotto del piano passante per $\tilde{\mathrm{A}}, \tilde{\mathrm{B}}, \tilde{\mathrm{C}}$. Questo, grazie al Lemma, si traduce in: la circonferenza passante per A, B e C non contiene alcun altro punto di $\mathcal{S}$ al suo interno. La proposizione è vera se e solo se i tre punti $A, \mathrm{B}, \mathrm{C}$ formano un triangolo della TD di $\mathcal{S}$.

- Dunque, la proiezione all'indietro di $\mathrm{LCH}(\tilde{\mathcal{S}})$ sul piano è la TD di $\mathcal{S}$. Un algoritmo che calcola il guscio convesso in E^3 serve a calcolare la TD in E^2.

Problemi collegati

- Non sorprende che la TD risolva gli stessi problemi del DV, essendone il duale.
- In particolare ALL NEAREST NEIGHBOURS si vede bene sulla TD grazie alla seguente proprietà:
- Le coppie di punti più vicini in $\mathcal{S}$ sono connessi da uno spigolo nella triangolazione di Delaunay (Dim. Per una coppia di punti più vicini passa una circonferenza $\mathcal{S}$-libera, altrimenti ci sarebbe un punto più vicino.)
- Inoltre la TD serve a risolvere in modo efficiente anche il seguente:
- ***Problema***: (MINIMO ALBERO EUCLIDEO DI RICOPRIMENTO, EMST) Dati n punti nel piano, costruire l'albero di minima lunghezza i cui vertici sono i punti dati.
- Un *albero* è un grafo aciclico massimale, ovvero un grafo privo di cicli nel quale l'aggiunta di uno spigolo qualunque crea un ciclo.
- Un ciclo in un grafo è un percorso in cui il primo vertice coincide con l'ultimo.

Minimo albero di ricoprimento

- Consideriamo il seguente problema, che generalizza il EMST precedente:
- ***Problema***: (Minimo albero di ricoprimento, MST) Dato un grafo connesso $G = (\mathcal{V}, \mathcal{E})$ in cui ad ogni arco è associato un costo, trovare un sottoinsieme degli spigoli che includa tutti i vertici, sia un albero e abbia costo minimo.
- EMST si ottiene da MST considerano il grafo completo che ha per vertici gli n punti dati e pesi pari alla distanza tra i vertici (grafo Euclideo).
- L'***algoritmo di Kruskal*** risolve MST con una strategia ***greedy***.
  1. Crea un insieme di alberi $\mathcal{A}$ in cui ogni vertice di $\mathcal{V}$ è un albero.
  2. Rimuovi lo spigolo e^* di costo minimo da $\mathcal{E}$
  3. Se e^* connette due alberi differenti aggiungilo ad $\mathcal{A}$ e fondi i due alberi in uno.
  4. Ripeti dal passo 2 finché $\mathcal{E}$ non è vuoto
- Alla fine $\mathcal{A}$ consiste di un solo albero, che è il MST.
- Al passo 3 il fatto che lo spigolo connetta due alberi distinti garantisce che non introduce un ciclo, quindi la nuova componente connessa è ancora un albero.

- ***Complessità*** Per ottenere lo spigolo di minor costo si possono ordinare preventivamente gli spigoli, con un costo $|\mathcal{E}| \log |\mathcal{E}|$.

  Controllare a quale albero appartenga un vertice (passo 3) costa $O(\log n)$ dove n è il numero di vertici dell'albero (usando una struttura dati opportuna). Il ciclo viene eseguito $O(|\mathcal{E}|)$ volte, quindi il costo totale è $O(|\mathcal{E}| \log |\mathcal{E}|) = O(|\mathcal{E}| \log |\mathcal{V}|)$.

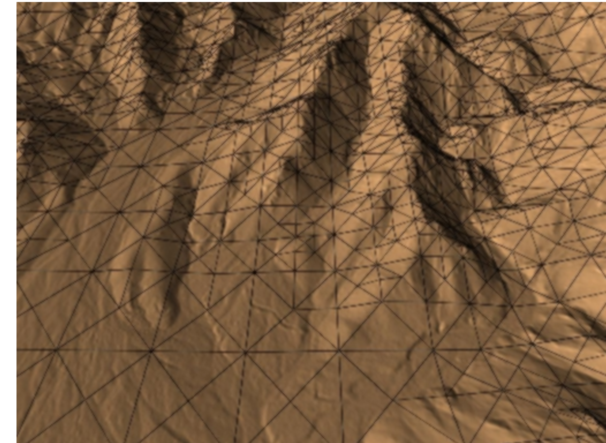
- Se si applica l'algoritmo di Kruskal a EMST (quindi al grafo completo con n vertici) il costo è $O(n^2 \log n)$.

- Tuttavia, è dimostrato il seguente:

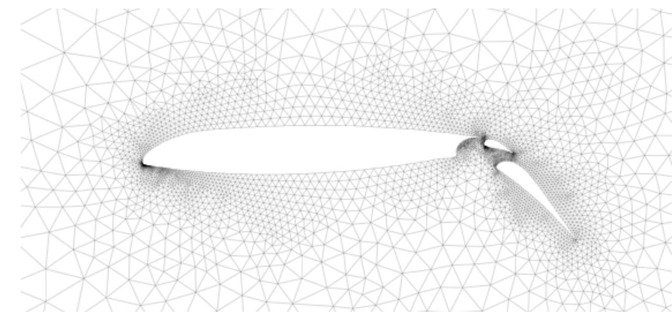
- ***Teorema***: Il minimo albero Euclideo di ricoprimento di un insieme di punti $\mathcal{S}$ è un sottografo della TD di $\mathcal{S}$.

- Grazie al Teorema si può prima calcolare la TD dei punti (costo $O(n \log n)$), e quindi impiegare l'algoritmo di Kruskal sulla TD, che ha $O(n)$ spigoli, con un costo $O(n \log n)$.

Applicazioni della TD



- Interpolazione numerica di dati bivariati campionati su un dominio discreto irregolare.
- 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.
- Per lo stesso motivo (evitare angoli troppo acuti) la TD è impiegata nella analisi agli elementi finiti.



Crediti

Queste diapositive sono basate su di un nucleo creato da Riccardo Giannitrapani (suo è anche lo stile LaTex) e successivamente rielaborato da me, per il corso di Grafica al Calcolatore. Molte spiegazioni seguono da vicino [5] e [6]. Alcune figure sono prese da [6] (mi riprometto per il futuro di sostituirle). La parte sulla triangolazione di Delaunay segue [7]. In generale, i testi di riferimento sono [1] e [2].

Copyright

Con l'esclusione delle figure sopra citate, quest'opera è pubblicata con la licenza Creative Commons Attribuzione-NonCommerciale-StessaLicenza. Per visionare una copia di questa licenza visita http://creativecommons.org/licenses/by-nc-sa/2.0/deed.it oppure spedisci una lettera a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Riferimenti bibliografici

[1] F. Preparata, M. Shamos. *Computational Geometry. An Introduction.* Springer, 1985.

[2] M. de Berg et al. *Computational Geometry. Algorithms and applications.* Springer, 2nd edition, 1999.

[3] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 2nd edition, 1998.

[4] H. Samet, *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[5] L. De Floriani, P. Magillo. *Dispense di I.U.M. Modellazione geometrica.* http://www.disi.unige.it/person/MagilloP/GEOMOD02/

[6] D. Mount. *CMSC 754 Lecture Notes.* ftp://ftp.cs.umd.edu/pub/faculty/mount/754/754lects.ps.gz

[7] J. R. Shewchuk. *Lecture Notes on Delaunay Mesh Generation.* http://www.cs.berkeley.edu/ jrs/meshpapers/delnotes.ps.gz

Indice dei problemi

All Nearest Neighbours, 124, 140, 158

Closest Pair, 124, 125
Convex Hull, 31, 33, 38, 41, 44, 45, 97
Convex Polygon Inclusion, 100, 103
Convex Polygon Intersection, 69, 70, 72
Convex Polyhedron Inclusion, 106

Extreme Points, 31, 45

General Position Test, 97

Half-plane Intersection, 73, 97, 98
Half-space Intersection, 75

Line-Segment Intersection, 54, 56, 62, 92
Line-Segment Intersection Test, 62, 65, 66
Lines Arrangement, 92, 93
Loci of Proximity, 124, 127, 132, 133

Minimo albero di ricoprimento, 159
Minimo albero Euclideo di ricoprimento, 158

Nearest Neighbour Search, 140

Planar Point Location, 107, 108, 140
Polygon Inclusion, 66, 100, 101, 107
Polygon Intersection, 67, 68
Polygon Intersection Test, 66
Polygon Triangulation, 24, 25
Polyhedron Inclusion, 106

Range Search - Count, 112, 120–123
Range Search - Report, 112, 120, 121, 123

Simplicity Test, 65

Triangulation, 87, 88

Upper/Lower envelope, 97, 98