Report PCS

Aldo Sambo, Maddalena Ghiotti, Emilio Zorzi

July 2023

1 Introduzione: concetti preliminari

Nel corso della trattazione considereremo un insieme $\mathcal{P} := \{p_1, p_2, ..., p_n\}$, dove $p_i := (x_i, y_i) \in \mathbb{R}^2$.

Definizione di Insieme Convesso

Un insieme $C \subseteq \mathbb{R}^2$ si dice convesso se:

$$\forall p_1, p_2 \in C, \;\; p_1 + (1-t)p_2 \in C \quad \forall t \in [0,1] \tag{1}$$

In altre parole, dati due punti dell'insieme, il segmento che li congiunge è interamente contenuto all'interno dell'insieme.

Definizione di Inviluppo Convesso (Convex Hull)

L'inviluppo convesso $\mathcal{H}(\mathcal{P})$ di un insieme di punti $\mathcal{P}$ è definito come l'intersezione di tutti gli insiemi convessi che lo contengono. Si può esprimere nel seguente modo:

$$\mathcal{H} := \{\sum_{i=1}^{n} \lambda_i p_i\} \tag{2}$$

con $\lambda_i \geq 0 \;\; \forall i \;\; \wedge \;\; \sum_{i=1}^{n} \lambda_i = 1$

Definizione di Triangolazione (di un insieme di punti $\mathcal{P}$)

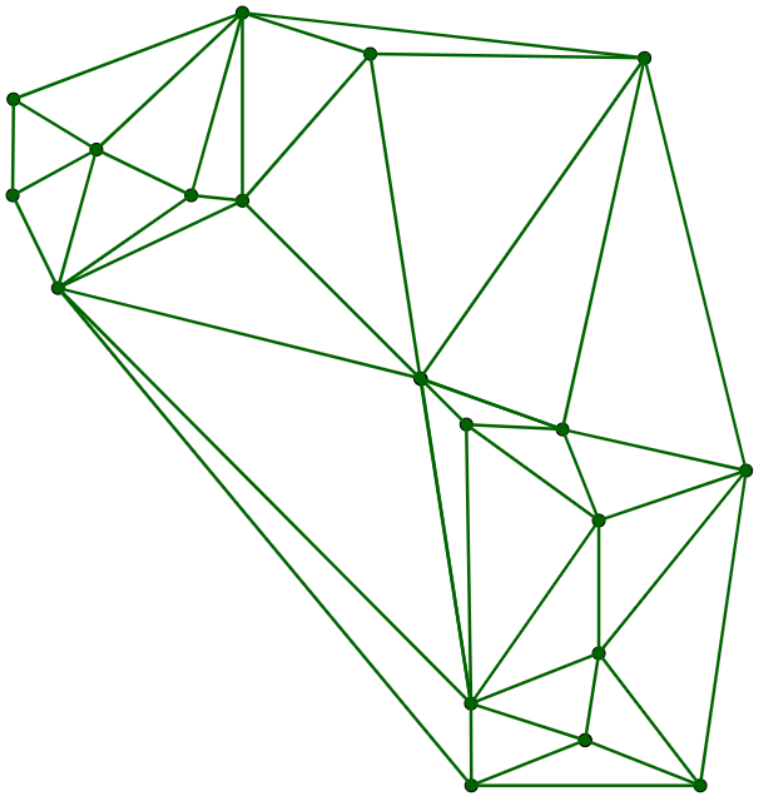
La triangolazione $\mathcal{T}(\mathcal{P})$ di un insieme di punti $\mathcal{P}$ consiste in una partizione dell'inviluppo convesso dei punti di $\mathcal{P}$ in triangoli i cui vertici siano tali punti, in modo tale che tali triangoli non contengano altri punti di $\mathcal{P}$.

Definizione di Triangolazione di Delaunay

Una triangolazione di $\mathcal{P}$ è detta di Delaunay se il circumcerchio[1] D di ogni triangolo della triangolazione è tale che D° non contiene alcun punto di $\mathcal{P}$.

[1] Il circumcerchio di un triangolo è la circonferenza passante per i tre vertici del triangolo stesso

Si dimostra che dato un insieme di punti $\mathcal{P}$ che non siano tutti allineati è sempre possibile trovare una triangolazione di Delaunay.



Esempio di una triangolazione di Delaunay

2 Introduzione all'Algoritmo di Delaunay

Abbiamo realizzato un algoritmo al fine di realizzare una triangolazione di Delaunay a partire da un insieme di punti $\mathcal{P}$.
Il nostro algoritmo è di stampo incrementale. Ciò significa che non considera immediatamente l'intero insieme $\mathcal{P}$, ma che ha origine da una prima triangolazione relativa a un insieme $\mathcal{P}_1 \subseteq \mathcal{P}$, formato da tre punti scelti ad hoc (di fatto un triangolo). Successivamente, ad esso aggiungiamo iterativamente tutti i punti in $\mathcal{P} \setminus \mathcal{P}_1$, in modo che ad ogni iterazione abbiamo un insieme $\mathcal{P}_i = \mathcal{P}_{i-1} \cup \{p_i\}$.

L'algoritmo è composto sostanzialmente da tre parti:

1) *Snake*
   Ricerca dei quattro punti iniziali a partire dai quali inizializzare l'algoritmo. In particolare, i primi tre punti costituiranno $\mathcal{P}_1$.

2) *Triangulation expansion*
   Aggiunta di un punto $p \in \mathcal{P} \setminus \mathcal{P}_i$ (con $i = 1$ alla prima iterazione) a $\mathcal{P}_\rangle$ e creazione di una nuova triangolazione $\mathcal{T}(\mathcal{P}_i \cup p)$.

3) *Flip propagation*
   Controllo del soddisfacimento della proprietà di Delaunay da parte della nuova triangolazione e sua modifica nel caso sia necessario. Al termine di questa sezione si pone $i = i + 1$.
   Se $i = n$, l'algoritmo termina.
   Se $i < n$, si ritorna al punto 2).

L'operazione detta *Snake* prevede la creazione di una griglia (*Grid*), come vedremo in seguito. Il suo obiettivo è quello di massimizzare il numero di punti interni presenti nel primo triangolo considerato (coincidente con la triangolazione di $\mathcal{P}_1$).

La *Triangulation Expansion* può avvenire in due modi diversi:

- il punto in $\mathcal{P} \setminus \mathcal{P}_i$ è interno alla triangolazione di $\mathcal{T}(\mathcal{P}_i)$. In tal caso, a seconda che esso appartenga al lato di un triangolo di $\mathcal{T}(\mathcal{P}_i)$ o meno, $\mathcal{T}(\mathcal{P}_{i+1})$ presenterà tre o due triangoli in più rispetto a $\mathcal{T}(\mathcal{P}_i)$;
- il punto in $\mathcal{P} \setminus \mathcal{P}_i$ è esterno alla triangolazione di $\mathcal{T}(\mathcal{P}_i)$. In tal caso si sfrutte l'inviluppo convesso $\mathcal{H}(\mathcal{P}_i)$ per comprendere quali triangoli aggiungere a $\mathcal{T}(\mathcal{P}_i)$ per ottenere $\mathcal{T}(\mathcal{P}_{i+1})$.

La *Flip Propagation* ci assicura, in particolare, che per ogni nuovo triangolo in $\mathcal{T}(\mathcal{P}_{i+1})$, la proprietà di Delaunay sia soddisfatta rispetto ai triangoli ad esso adiacenti. In caso contrario, avviene la procedura detta di *Flip*, che deve essere *propagata* lungo tutta la triangolazione.

Queste tre parti dell'algoritmo verranno adesso trattate più dettagliatamente.

3 Operazione *Snake*

Motivazione

Ci siamo domandati quale tra le seguenti scelte potesse contribuire a una maggior efficienza:

- *Da Destra a Sinistra*
  Il nuovo punto da aggiungere di volta in volta è sempre esterno alla triangolazione già creata o sul bordo di essa. Entra sempre in gioco l'inviluppo convesso $\mathcal{H}(\mathcal{P}_i)$;

- *Random*
  I tre punti iniziali sono scelti in maniera randomica;

- *Snake* (idea nostra)
  i tre punti iniziali sono scelti in modo da massimizzare *empiricamente* il numero di punti presente nella triangolazione $\mathcal{T}(\mathcal{P}_1)$ (dunque nel primo triangolo).

La nostra supposizione teorica (poi confermata dalla pratica) era che lo *Snake* avrebbe dovuto funzionare meglio nel nostro algoritmo, soprattutto per un gran numero di punti, per due principali motivazioni:

1) Tutta la letteratura da noi trovata a riguardo della triangolazione di Delaunay presenta la *costruzione* di un triangolo a partire da tre nuovi punti *fittizi* in modo tale che tutti i punti di $\mathcal{P}$ siano contenuti dentro tale triangolo;

2) Abbiamo trovato un articolo scientifico in cui si dimostra esplicitamente che un algoritmo incrementale per la triangolazione di Delaunay basato sull'inserimento randomico di un punto sempre *interno* ha un costo computazionale pari a $O(nlogn)$ e che la creazione di una triangolazione di Delaunay ha un costo computazionale $\Omega(nlogn)$.

Per trovare i primi tre punti, abbiamo dunque costruito il seguente algoritmo.
Come prima cosa, costruiamo una griglia (*Grid*) rettangolare di $(\lfloor\sqrt{n}\rfloor)^2$ rettangoli in modo tale che ogni punto sia all'interno di essa. Ciò prevede due passaggi:

- Individuo i punti aventi ordinata massima e ordinata minima (y_{min}ey_{max}), assieme ai punti aventi ascissa minima e ascissa massima (x_{min}ex_{max}). Al fine di ottenerli, itero lungo tutto il vettore effettuando ogni volta quattro controlli (con eventuali sostituzioni), per un costo computazionale $O(n)$

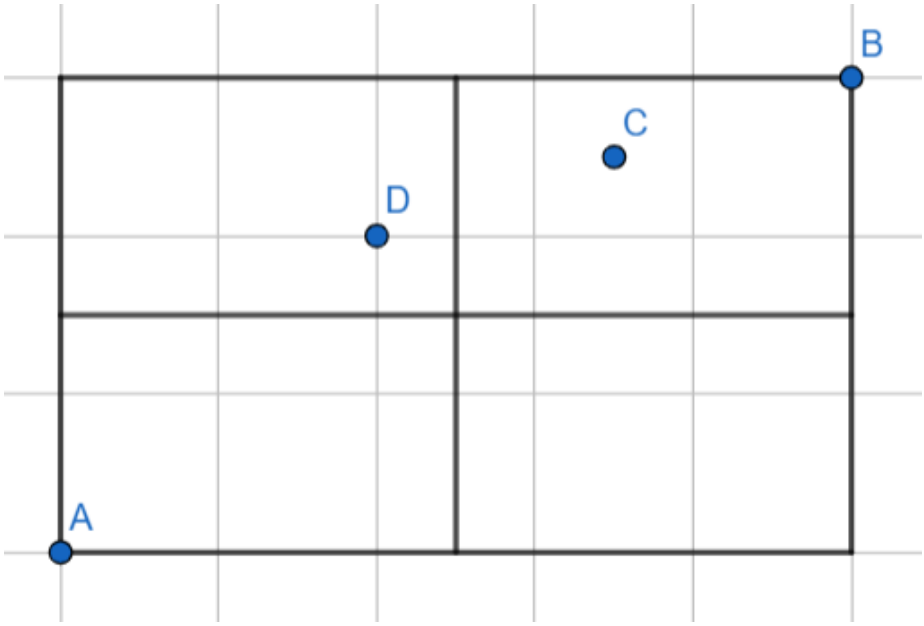
- Divido la parte di piano contenuta tra le rette $y = y_{min}$ e $y = y_{max}$ in $\lfloor\sqrt{n}\rfloor$ parti, uniformemente, mediante l'operazione:

$$\frac{(y_{max} - y_{min})}{\lfloor\sqrt{n}\rfloor} \tag{3}$$

- Divido la parte di piano contenuta tra le rette $x = x_{min}$ e $x = x_{max}$ in $\lfloor\sqrt{n}\rfloor$ parti, uniformemente, mediante l'operazione:

$$\frac{(x_{max} - x_{min})}{\lfloor\sqrt{n}\rfloor} \tag{4}$$

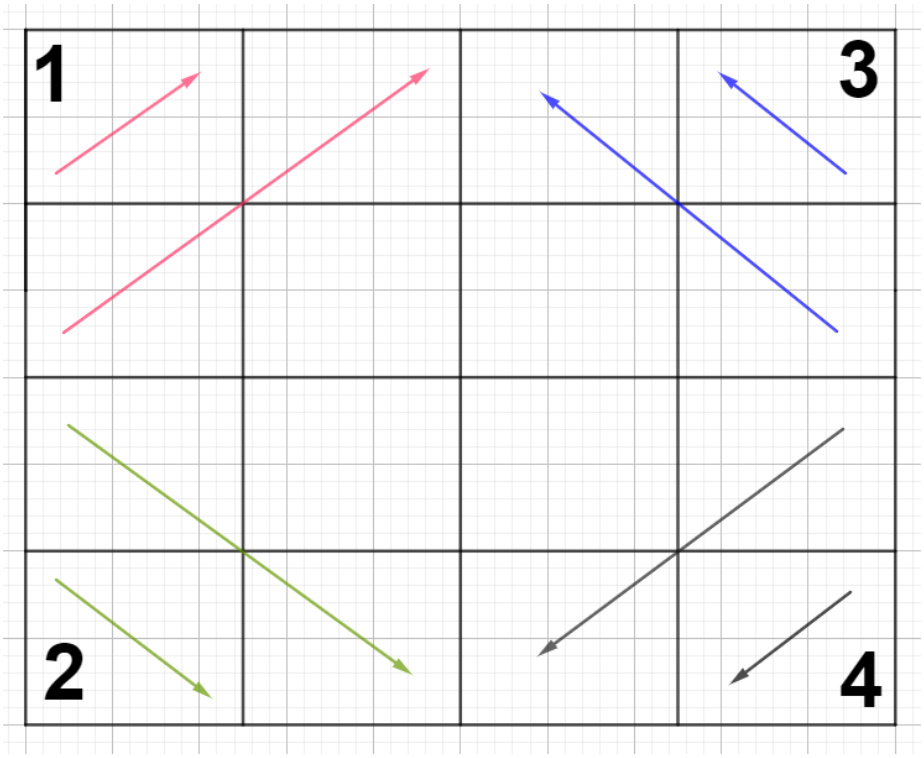
In tal modo ottengo una griglia (in generale) rettangolare formata (in generale) da rettangoli, in quanto $|x_{max}{-}x_{min}|$ non è necessariamente uguale a $|y_{max}{-}y_{min}|$. .



Esempio semplice di una costruzione di una grid dati 4 punti

In secondo luogo, eseguiamo una scansione della griglia al fine di ricercare i primi quattro punti da considerare nel nostro algoritmo. Al fine di illustrare tale procedimento, considereremo la griglia (che chiameremo G) come una matrice $\lfloor\sqrt{n}\rfloor \times \lfloor\sqrt{n}\rfloor$. Ad esempio, *il primo elemento in alto a sinistra* verrà indicato come $G(0, 0)$.
Per scegliere il primo punto da inserire in $\mathcal{P}_1$, considero il rettangolo della griglia G(0,0). Se vi è un punto in tale rettangolo, lo inserisco in $\mathcal{P}_1$. Altrimenti, fisso $i = 1$ e ripeto il procedimento su $G(j, k)$, in cui $j + k = i$ e $j = i$. Se non trovo un punto neanche dentro tale rettangolo, fisso $j = j - 1$ e $k = k + 1$. Se nuovamente non riesco a trovare un punto dentro $G(j, k)$, fisso $i = i + 1$ e ripeto tutto il procedimento, fino a quando non individuo un rettangolo che presenti effettivamente al suo interno un punto. Il fatto che incontrerò almeno un rettangolo con un punto al suo interno è assicurato dall antura del problema.

Per trovare rispettivamente il secondo, il terzo e il quarto punto l'idea è simile, e consiste nel partire dagli *angoli* della matrice e procedere in diagonale fino a quando non si identifica un punto, secondo il seguente schema:



Costo computazionale scan

Tale operazione ha un costo computazionale pari a $O(4n) = O(n)$, in cui il bound è stretto, poiché ad ogni iterazione visiterò strettamente meno di $(\lfloor\sqrt{n}\rfloor)^2 < n$ rettangoli.

Collisione tra rettangoli

Nel caso in una delle quattro *scansioni* della griglia incontrassimo un rettangolo in cui è già stata dichiarata la presenza di un punto (con conseguente scelta) in una *scansione* precedente, affermeremmo che il nostro metodo ha fallito (onde evitare la costruzione di triangoli degeneri), e sceglieremmo i quattro punti iniziali casualmente (facendo attenzione che i primi tre non siano allineati).

Punti allineati

Nel caso in cui l'algoritmo restituisse tre punti allineati, dichiareremmo come nella sezione precedente che il metodo ha fallito, e sceglieremmo i primi quattro punti casualmente (come prima, prestando attenzione al fatto che i primi tre punti non siano allineati).

Conclusioni sullo *Snake*

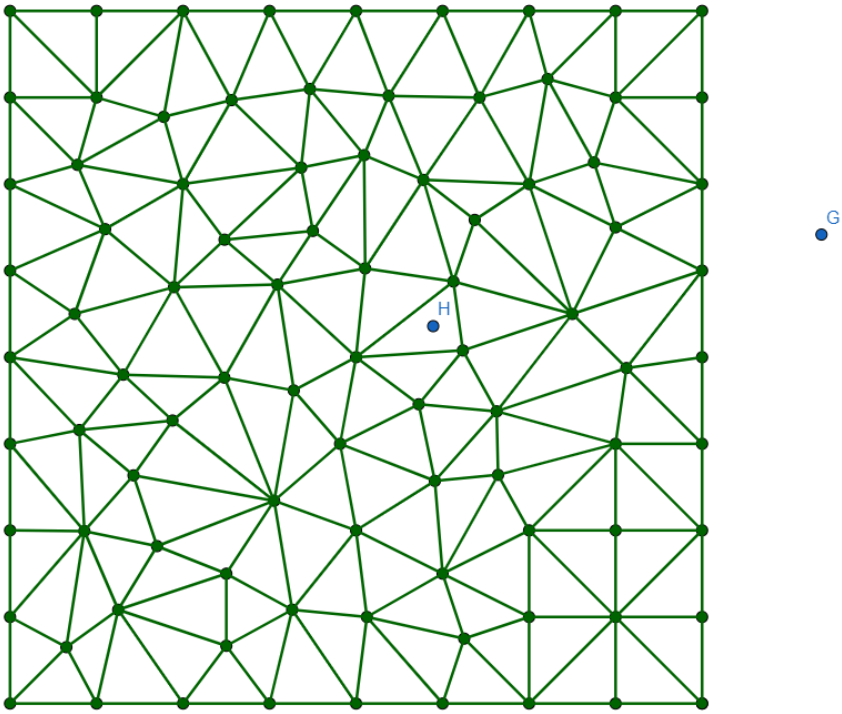
Il costo complessivo dell'operazione *Snake* è $O(n)+O(n) = O(n)$. Non è sempre possibile applicare lo *Snake* in tutti casi: a volte è necessario comunque generare i primi quattro punti casualmente.

4 Procedura di *Triangulation expansion*

Idee di base

Data la natura iterativa del nostro approccio, questa procedura è definita per un generico punto $p \in \mathcal{P} \setminus \mathcal{P}_{i-1}$, e dunque per l'$i$-esimo passo dell'algoritmo.

Come accennato nella sezione *Introduzione all'Algoritmo di Delaunay* l'aggiunta di tale punto avviene in modo diverso se $p \in \mathcal{T}(\mathcal{P}_{i-1})$ oppure se $p \notin \mathcal{T}(\mathcal{P}_{i-1})$.



Esempio: il punto G è esterno a $\mathcal{T}(\mathcal{P}_{i-1})$, mentre il punto H è interno a $\mathcal{T}(\mathcal{P}_{i-1})$

Riconoscimento di un punto esterno/interno

Al fine di riconoscere nella pratica la posizione del punto p rispetto alla triangolazione $\mathcal{T}(\mathcal{P}_{i-1})$, ricorro all'inviluppo convesso $\mathcal{H}(\mathcal{P}_i)$. Considero infatti $\overline{\mathcal{H}(\mathcal{P}_i)} \setminus (\mathcal{H}(\mathcal{P}_i))^\circ$, ossia la frontiera dell'inviluppo convesso, costituita dai lati

sul bordo. Per ogni lato $\vec{xy}$ di tale frontiera, calcolo il prodotto vettoriale

$$\vec{yp} \times \vec{xy} := ||\vec{yp}||\,||\vec{xy}||sen(\theta)\mathbf{k} \tag{5}$$

dove θ è l'angolo compreso tra $\vec{yp}$ e $\vec{xy}$. Il segno di tale prodotto vettoriale ci restituisce la posizione del punto p rispetto al lato della frontiera dell'inviluppo convesso. Ci sono due opzioni:

- $\exists \vec{xy},\ \overline{xy} \in \mathcal{H}(\mathcal{P}_i)\ :\ \vec{yp} \times \vec{xy} < 0$. In questo caso il punto si trova all'esterno dell'inviluppo convesso, e dunque all'esterno della triangolazione;
- $\forall \vec{xy},\ \overline{xy} \in \mathcal{H}(\mathcal{P}_i)\ :\ \vec{yp} \times \vec{xy} >= 0$. In questo caso il punto si trova all'interno dell'inviluppo convesso, e dunque all'interno della triangolazione.

Espansione: punto interno

Se il punto è interno alla triangolazione, si presentano due sottocasi rilevanti, ossia:

- $\exists \vec{xy},\ \overline{xy} \in \mathcal{H}(\mathcal{P}_i)\ :\ \vec{yp} \times \vec{xy} = 0$, ossia il punto è sul lato di un triangolo $T_j \in \mathcal{T}(\mathcal{P}_{i-1})$ (**punto sul bordo dell'inviluppo convesso**);
- $\forall \vec{xy},\ \overline{xy} \in \mathcal{H}(\mathcal{P}_i)\ :\ \vec{yp} \times \vec{xy} > 0$, ossia il punto non è sul lato di un triangolo $T_j \in \mathcal{T}(\mathcal{P}_{i-1})$ (**punto strettamente interno**).

Punto strettamente interno

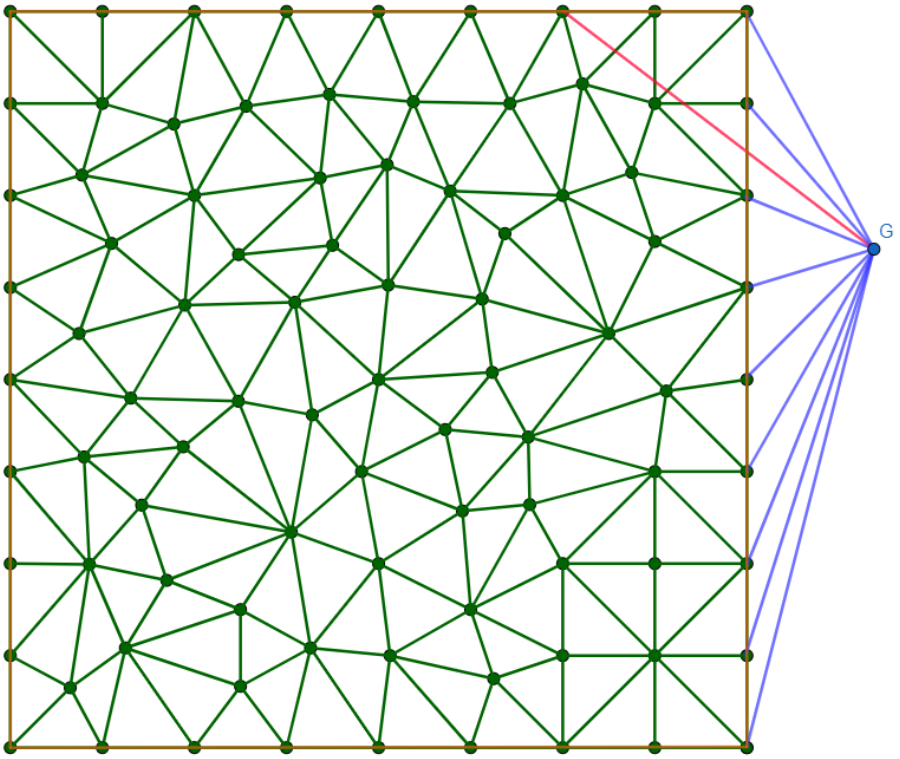
Se il punto è strettamente interno alla triangolazione, l'operazione di *Triangulation expansion* consisterà nel collegamento dei tre vertici del triangolo in cui è contenuto tale punto con il punto stesso.
Vi è tuttavia un'importante eccezione: se esso infatti si trova su un lato dell'intersezione di tutti i triangoli in cui è contenuto, occorrerà collegarlo con entrambi i vertici dei triangoli che condividono il lato su cui si trova e che non siano estremi di tale lato.

Punto sul bordo

In questo caso, il punto è sulla frontiera dell'inviluppo convesso e per l'operazione di *Triangulation expansion* sarà sufficiente collegare i due vertici del triangolo a cui appartiene con esso.

Espansione: punto esterno

Se il punto è esterno alla triangolazione $\mathcal{T}(\mathcal{P}_{i-1})$, l'operazione è più complessa e richiede di iterare sui lati della frontiera dell'inviluppo convesso al fine di trovare i punti di tale frontiera da collegare con il nuovo punto.
Questa procedura richiede ancora una volta di valutare i prodotti vettoriali $\vec{yp} \times \vec{xy}$, con $\overline{xy} \in \mathcal{H}(\mathcal{P}_i)$, al fine di comprendere rispetto a quali punti un eventuale collegamento $\overline{yp}$ sarebbe tale che $\overline{yp} \cap \overline{xy} \neq 0$.



Esempio: vogliamo collegare il punto G evitando intersezioni con i lati della frontiera dell'inviluppo convesso (in arancione). In blu i collegamenti che siamo interessati a fare. Vogliamo evitare la costruzione del lato di colore rosso.

Implementazione nella pratica

L'implementazione nella pratica sfrutta due principali idee:

- **la costruzione di un oggetto "convexHullElem"**, composto da un puntatore a punto, un puntatore a triangolo e da due puntatori ad altri "convexHullElem". Racchiude nella sua definizone le finalità per cui l'abbiamo costruito: ci interessa, dati i punti della frontiera dell'inviluppo convesso, conoscere il punto successivo e il precedente, secondo un qualche ordine, in modo da poter iterare sui punti della frontiera in modo efficace. Siamo poi interessati ad associarvi un triangolo per necessità che saranno più chiare più avanti;
- **La costruzione di un sistema di puntatori tra triangoli**, in modo che la ricerca del triangolo "più piccolo" in cui è contenuto un punto (formalmente la ricerca dell'intersezione di tutti i triangoli che lo contengono)

sia la più veloce possibile. Il suo utilizzo, come vedremo più avanti, non si limiterà a questo.

Ora esponiamo in che modo esattamente abbiamo costruito il sistema di puntatori tra triangoli e come ciò si ricollega al caso del punto esterno.

Triangoli guida

Cosa succede nel caso del punto esterno

Dopo la scelta dei tre punti iniziali secondo l'operazione *Snake*, costituiamo con essi la prima triangolazione e inseriamo tale triangolo nel vettore di triangoli "guideTriangles" (per la precisione si tratta di un vettore di puntatori a triangolo, ma a livello prettamente teorico il significato ultimo è lo stesso). Ciascun triangolo in tale vettore è un *triangolo guida*. Ogni volta che verrà aggiunto un punto esterno alla triangolazione, si formeranno m nuovi triangoli guida in seguito al collegamento di tale punto ai punti della frontiera dell'inviluppo convesso in modo da non creare intersezioni.

Aggiornamento del sistema di puntatori

Ogni volta che un punto interno alla triangolazione (e quindi necessariamente a qualche triangolo guida) viene aggiunto, il sistema si aggiorna cosicché il triangolo guida che viene suddiviso in *sottotriangoli* presenti come attributo un vettore di puntatori contenente l'indirizzo di memoria di tali *sottotriangoli*.

Ricerca del triangolo "più piccolo" mediante il sistema di puntatori

Il vettore di triangoli guida costiuirà lungo tutto il corso dell'algoritmo il punto di partenza del sistema di puntatori tra triangoli, nel senso che, se il punto è interno, per prima cosa si controllerà in quale dei triangoli guida esso è contenuto. In seguito, mediante il sistema di puntatori, si ricercherà il triangolo più "piccolo" in cui il punto è contenuto, il quale, per costruzione, non punterà ad alcun altro triangolo.

Come comprendere nella pratica se un punto è interno a un triangolo

Per comprendere se un punto è interno al triangolo, sul bordo di esso, o esterno ad esso, si agisce nello stesso modo in cui si stabilisce se un punto è interno e esterno all'intera triangolazione (si può dire che il triangolo è una *sottotriangolazione* relativa a tre punti di $\mathcal{P}$). In questo caso i lati della frontiera dell'inviluppo convesso non sono altro che i lati del triangolo stesso.

Conclusioni e tentativo di studio del costo computazionale

Nel caso di aggiunta di un punto interno, il costo computazionale è dato dalla ricerca sul vettore di triangoli guida e poi nel sistema di puntatori a triangoli unita al tempo costante necessario alla suddivisione in sottotriangoli. Il numero di triangoli guida dipenderà dall'efficacia nel racchiudere nella prima triangolazione considerata il maggior numero di punti possibili.
Nel caso riuscissimo a contenere tutti i punti nel primo triangolo, il costo computazionale sarebbe di $O(1)$ per la ricerca del triangolo guida (c'è solo un triangolo guida) e all'incirca $O(logn)$ per la ricerca del punto interno (al massimo un triangolo punta ad altri tre triangoli nel sistema di puntatori). Ripetendo n volte (il numero dei punti) tale operazione, otteniamo un costo computazionale pari a $O(nlogn)$.
Nel caso di aggiunta di un punto esterno, la situazione diventa molto più complicata: c'è infatti da considerare il comportamento della frontiera dell'inviluppo convesso all'aumentare di n. Da quanto emerge da uno studio empirico, il numero di elementi nella frontiera dell'inviluppo convesso all'aumentare di n ha un comportamento simile a $logn$. Si può dunque stimare che se tutti i punti aggiunti fossero esterni il costo totale delle operazioni sarebbe comunque $O(nlogn)$.

5 Flip propagation

Introduzione e idee teoriche

Ogni volta che aggiungiamo un nuovo punto alla nostra mesh, nel passaggio da $\mathcal{T}(\mathcal{P}_{i-1})$ a $\mathcal{T}(\mathcal{P}_i)$, dobbiamo assicurarci che la nuova triangolazione sia *Delaunay*.
Dal momento che l'unica differenza a livello di punti rispetto al passaggio precedente è data dalla presenza del nuovo punto, vogliamo verificare che:

- tutti i circumcerchi dei triangoli della triangolazione $\mathcal{T}(\mathcal{P}_{i-1})$ (meno eventualmente i triangoli che sono stati suddivisi in *sottotriangoli*) non contengano il nuovo punto;
- i circumcerchi dei *nuovi*[2] triangoli della triangolazione $\mathcal{T}(\mathcal{P}_i)$ non contengano punti di $\mathcal{P}_{i-1}$.

Algoritmo vero e proprio

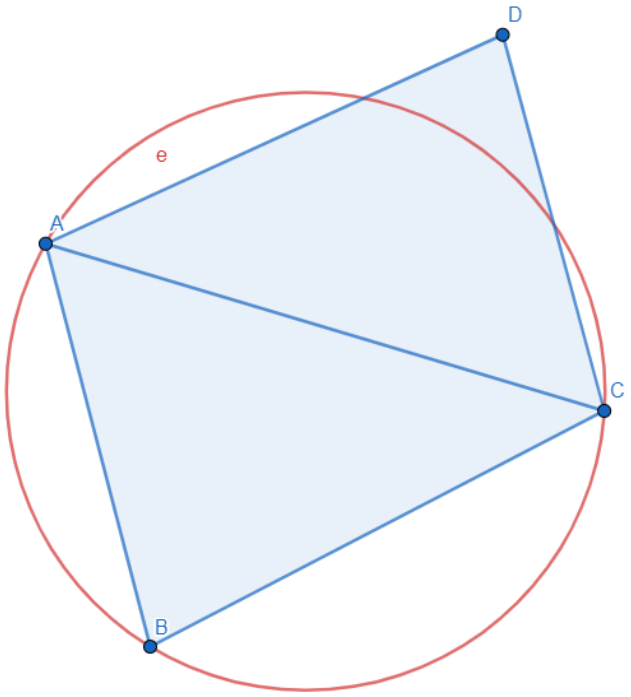
Per assicurarci che entrambi i punti citati nella sezione precedente siano verificati, è necessario ricorrere a un algoritmo ricorsivo, così strutturato:

1) considero il vettore di (puntatori a) triangoli che ci restituisce la sezione di *Triangulation Expansion*;
2) itero su tale vettore; in tal modo, alla i-esima iterazione staremo considerano un (puntatore a) triangolo specifico;
3) considero i triangoli adiacenti a tale triangolo nella triangolazione, non necessariamente *Delaunay* e derivante dall'operazione di *Triangulation expansion*, $\mathcal{T}'(\mathcal{P}_i)$. In particolare, mi interesso alla triangolazione $\mathcal{T}'(\mathcal{P}_i^{loc})$ dei punti $\mathcal{P}_i^{loc}$ che sono i vertici del triangolo specifico considerato al punto precedente e dei triangoli ad esso adiacenti;
4) Verifico se il circumcerchio dei triangoli adiacenti contiene il nuovo punto aggiuntosi. In caso affermativo, modifico $\mathcal{T}'(\mathcal{P}_i^{loc})$ attraverso l'operazione detta di *Flip*. Ottengo dunque la triangolazione $\mathcal{T}(\mathcal{P}_i^{loc})$, che è *Delaunay* per $\mathcal{P}_i^{loc}$;
5) nel caso sia stata applicata la procedura di *Flip*, devo controllare che i triangoli in $\mathcal{T}(\mathcal{P}_i^{loc}) \setminus \mathcal{T}'(\mathcal{P}_i^{loc})$ soddisfino l'ipotesi di Delaunay rispetto a tutti i punti in $\mathcal{P}_i \setminus \mathcal{P}_i^{loc}$. Ciò significa che devo ritornare ricorsivamente al punto 2), considerando un vettore di (puntatori a) triangoli contenente tutti i triangoli in $\mathcal{T}(\mathcal{P}_i^{loc}) \setminus \mathcal{T}'(\mathcal{P}_i^{loc})$

Nell'implementazione pratica, il punto 3) è ripetuto iterativamente per un numero di volte pari al numero di triangoli adiacenti, e ogni volta $\mathcal{P}_i^{loc}$ è semplicemente costituito ogni volta dai quattro punti costituenti i due triangoli

[2] triangoli che sono presenti in $\mathcal{T}(\mathcal{P}_i) \setminus \mathcal{T}(\mathcal{P}_{i-1})$

adiacenti. Il significato resta il medesimo. In particolare, sarà sufficiente ogni volta verificare che il circumcerchio passante per il triangolo adiacente contenga o meno il nuovo punto aggiunto.



Esempio: Nella figura, due triangoli che soddisfano l'ipotesi di Delaunay.

Nota

Si dimostra che per controllare l'ipotesi di Delaunay tra due triangoli sia sufficiente controllare che la circonferenza circoscritta al primo non contenga il vertice *opposto* (nel senso di non comune tra i due triangoli) del secondo. Non è necessario ripetere l'operazione scambiando primo e secondo.

Verifica dell'ipotesi di Delaunay su una coppia di triangoli

Al fine di verificare che due triangoli adiacenti verifichino l'ipotesi di Delaunay, è sufficiente controllare la seguente condizione:

$$\begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix}$$

$$= \begin{vmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 \end{vmatrix} > 0$$

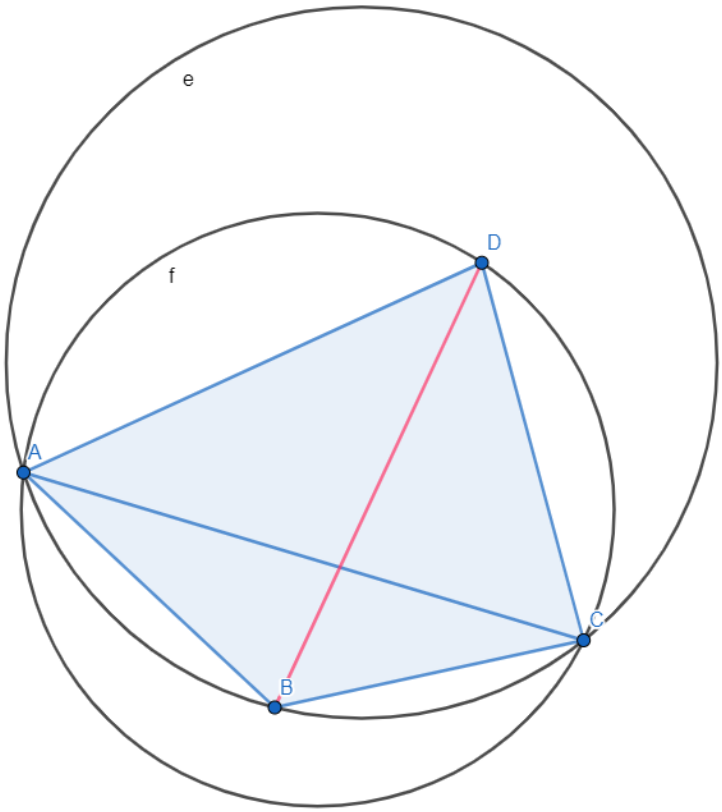
dove A, B, C sono i tre vertici del triangolo rispetto al quale consideriamo il circumcerchio e D è il punto che vogliamo verificare essere dentro il circumcerchio o meno. Nel caso sia soddisfatta, il punto D giace effettivamente all'interno del cirumcentro avente come vertici A, B, C ed è necessario procedere all'operazione di *Flip*.

Operazione di *Flip*

Dati due triangoli che non soddisfano l'ipotesi di Delaunay, l'operazione di *Flip* consiste nella modifica di un lato della triangolazione. Detti A, B, C i vertici del primo triangolo e B, C, D i vertici del secondo, il *Flip* produce due nuovi triangoli ADB e ADC.
La correttezza di questa operazione è immediatamente compresa nel momento in cui si riformula l'ipotesi di Delaunay nel senso degli angoli opposti dei due triangoli adiacenti: essi soddisfano l'ipotesi di Delaunay se la somma degli angoli opposti dei triangoli adiacenti è minore di π. Nel caso sia maggiore di tale valore, ricordando che la somma degli angoli interni a un quadrilatero è 2π, ci si rende immediatamente conto che la somma degli altri angoli del quadrilatero A, B, C, D deve essere minore di π, il che giustifica tutta l'operazione.

Sistema di puntatori a triangolo post-*Flip*

Ogni volta che avviene un *Flip* il sistema di puntatori viene aggiornato nel seguente modo: entrambi i nuovi triangoli derivanti dall'operazione sono puntati dai triangoli dai quali sono stati originati.
Mediante questa aggiunta, notiamo che, per costruzione, i triangoli che non punteranno a nessun altro triangolo della triangolazione saranno i triangoli della



Esempio di *Flip*: AC viene sostituito con BD

mesh più aggiornata, dove con mesh si intende l'insieme dei triangoli nella triangolazione. Una volta terminata l'inserzione di nuovi punti, saranno dunque i triangoli di questo tipo a fornirci la triangolazione di Delaunay.
Il sistema di puntatori a triangoli non è dunque propriamente un albero, poiché effettivamente dopo un flip abbiamo che due radici puntano a due medesime foglie. Si tratta più propriamente, dunque, di un grafo. Nell'esempio sopra riportato, ad esempio, i triangoli ABC e ACD punteranno ai triangoli post-*Flip* ABD e BCD.

Come ricavare la triangolazione finale

Notiamo che, per costruzione, i triangoli che non punteranno a nessun altro triangolo della triangolazione saranno i triangoli della *mesh* più aggiornata,

dove con mesh si intende l'insieme dei triangoli nella triangolazione. Una volta terminata l'inserzione di nuovi punti, saranno dunque i triangoli di questo tipo a fornirci la triangolazione di Delaunay.
Il sistema di puntatori non servirà dunque solamente a ritrovare il triangolo "più piccolo" in cui un punto è contenuto, ma anche a risalire alla triangolazione di Delaunay, sia nei passaggi intermedi che al termine dell'algoritmo.

Costo computazionale considerando i Flip

Nell'articolo [1] si dimostra, attraverso una dimostrazione piuttosto complicata che esula dalle finalità di questo report, che nel caso di inserimento di un punto *interno*, il numero medio di *Flip* necessari a passare da una triangolazione $\mathcal{T}'(\mathcal{P}_i)$ non necessariamente Delaunay a una triangolazione $\mathcal{T}(\mathcal{P}_i)$ Delaunay è $O(1)$, ossia costante.
Ciò significa che, nel caso il nostro algoritmo prevedesse soltanto inserimenti di punti interni, il costo computazionale totale sarebbe **in media** $O(nlogn)$, poiché per ogni punto degli n a disposizione la ricerca del triangolo "più piccolo" in cui si trova richiede **in media**[3] $O(logn)$ operazioni e i *Flip* richiedono $O(1)$ operazioni **in media**.
Tuttavia, il nostro algoritmo prevede anche la possibilità di aggiungere un punto esterno. In tal caso, è più difficile prevedere il numero di *Flip* necessari a ottenere una triangolazione di Delaunay. I risultati empirici che abbiamo ottenuto ci portano a pensare che tali operazioni non siano mediamente costanti.

[3] il fatto che il sistema di puntatori contenga anche lo storico dei *Flip* rende la ricerca del triangolo più piccolo **in media** $O(logn)$, e non nel caso peggiore

Bibliografia e Sitografia

[1] Leonidas J. Guibas, Donald E. Knuth and Micha Sharir (1992) *Randomized Incremental Construction of Delaunay and Voronoi Diagrams*, pubblicato su *Algorithmica*, Springer

[2] Bernd Gartner (2008) *Delaunay Triangulations*, ETH Lecture

[3] *Convex Hull*, Wolfram MathWorld

`https://mathworld.wolfram.com/ConvexHull.html`

[4] *Lecture 8 of Computational Geometry*, University of Arizona in Tucson, AZ

`https://www2.cs.arizona.edu/classes/cs437/fall11/Lecture8.pdf`