PROPRIETÀ DEL CONVEX HULL

- Dato un edge *pq* del CH, tutti i punti di P saranno nello stesso semipiano rispetto a *pq*.
- Seguendo il contorno di un CH in senso orario si compiono solo svolte verso destra.
- Il CH di un insieme di punti S è il poligono con il minimo perimetro (e viceversa).
- L'intersezione di due poligoni convessi è ancora un poligono convesso.
- Per un punto *q* esterno ad un poligono convesso C, si ha un unico punto *c* lungo C tale che la sua distanza da *q* (*cq*) sia minima.
- Per un punto *x* esterno ad un poligono convesso C, si ha una linea che separa *x* da C.

ALGORITMI DCEL

```
listIncidentVertices (vertex v)
	start = IncidentEdge(v)
	add(origin(twin(start)))
	edge = next(twin(start))
	while edge <> start do
		add(origin(twin(edge)))
		edge = next(twin(edge))
	end
```

```
listIncidentVertices (face v)
	start = OuterComponent(f)
	add(origin(start))
	edge = next(start)
	while edge <> start do
		add(origin(edge))
		edge = next(edge)
	end
	i = 1
	while InnerComponent(f)[i] <> NIL do
		start = InnerComponents(f)[i]
		add(origin(start))
		edge = next(start)
		while edge <> start do
			add(origin(edge))
			edge = next(edge)
		end
		i++
	end
```

listIncidentFace(vertex v)

```
start = IncidentEdge(v)
add(IncidentFace(start))
edge = next(twin(start))
while edge <> start do
      if IncidentFace(edge) is not present
            then add(IncidentFace(edge))
      edge = next(twin(edge))
end
```

listIncidentEdges (vertex v)

```
start = IncidentEdge(v)
add(start)
edge = next(twin(start))
while edge <> start do
      add(edge)
      edge = next(twin(edge))
end
```

listIncidentEdges (face f)

```
start = OuterComponent(f)
add(start)
edge = next(start)
while edge <> start do
      add(edge)
      edge = next(edge)
end
i = 1
while InnerComponent(f)[i] <> NIL do
      start = InnerComponents(f)[i]
      add(start)
      edge = next(start)
      while edge <> start do
            add(edge)
            edge = next(edge)
      end
      i++
end
```

```
listAdjacentFaces (face f)
	start = OuterComponent(f)
	add(IncidentFace(twin(start)))
	edge = next(start)
	while edge <> start do
		add(IncidentFace(twin(edge)))
		edge = next(edge)
	end
	i = 1
	while InnerComponent(f)[i] <> NIL do
		add(IncidenrFace(twin(Innercomponents(f)[i])))
		i++
	end
```

TRICOLORAZIONE DI UN POLIGONO SEMPLICE TRIANGOLATO

Input: *la triangolazione T di un poligono semplice P e il suo albero duale D*

Output: *ad ogni vertice di P è associato un colore ∈ {R,G,B}*

```
Tricolorazione(T,D)
	∀ w ∈ D do visited(w) ← False;
	scegli un vertice w ∈ D di grado uno;
	colora i vertici del triangolo T_w associato con R, G e B;
	visited(w) ← True
	∀ w' ∈ D adiacente a w ∧ visited(w) = False do
		DFS(w', D, T);
	end;


//Depth First Search
DFS(v, D, T)
	// v ∈ D vertice da cui proseguire la DFS
	// il triangolo τ ∈ T associato a v ha due vertici colorati
	visited(v) ← true;
	colora il vertice colorato di τ nell'unico modo possibile;
	∀ w ∈ D adiacente a v ∧ visited(w) = False do
		DFS(w, D, T);
	end;
```

COLORAZIONE DI UNA TRIANGOLAZIONE ESPRESSA CON UNA DCEL

```
colorize() {
	prendo un half-edge e dalla DCEL (il primo)
	origin(e).color = RED;
	origin(next(e)).color = green;
	colorizeTriang(e);
	colorizeTriang(twin(e));
}

colorizeTriang(Half-edge e) {
	//controllo se sono sul bordo esterno
	if (e != next(next(next(e)))) {
		//mi sposto all'interno del triangolo adiacente
		e = twin(e);
	}
	e1 = prev(e);
	e2 = next(e);
	vnew = origin(e1 );
	if (colored(vnew) = false) {
		vnew.color = getColor(e);
		colorizeTriang(twin(e1 ));
		colorizeTriang(twin(e2 ));
	}
}

getColor(Half-edge e) {
	//restituisce il terzo colore mancante nel triangolo
	for color ← Red, Green, Blue {
		if not [(origin(e) = color) or (origin(next(e)) = color )] {
			return color;
		}
	}
	return null;
}
```

VORONOI

Nella DCEL che salva le informazioni sulla partizione, si salvano anche i punti associandoli alla lista delle facce. Quindi sappiamo quale punto appartiene ad una determinata faccia.

Face	OuterComponent	InnerComponent	Point

Punti a distanza minima

Per trovare i punti a distanza minima possiamo attraversare gli edge della DCEL, trovare le due facce adiacenti e calcolare le distanze tra i punti contenuti.

Trovare punti a distanza minima: O(n)

```
Inizializzo il minimo Min = ∞
per ogni half-edge e non visitato nella lista degli half-edge {
// ricavo la distanza tra i punti appartenenti alle facce incidenti
// all'edge corrente ed al suo twin
p_i = Point(IncidentFace(e));
p_j = Point(IncidentFace(twin(e)));
// calcolo la distanza tra i punti
d = distance (p_i , p_j );
// aggiorno il minimo
if d < Min {
      Min = d;
}
marca (twin(e) come visitato
```

Siccome ad un edge del diagramma corrisponde un edge della triangolazione di Delaunay e gli edge della triangolazione soddisfano la proprietà del massimo cerchio vuoto, allora non ci sono punti più vicini di quelli considerati (appartenenti a due facce adiacenti nel diagramma di Voronoi).

Distanza minima tra ogni punto p ∈ P ed un altro punto più vicino

Nell'algoritmo di costruzione del diagramma di Voronoi si aggiungono i seguenti passi ogni volta che viene aggiunto un edge:

```
d = distance (pi , pj );
if d < minvect[i] {
      minvect[i] = d;
      closestpoint[i] = j;
}
if d < minvect[j] {
      minvect[j] = d;
      closestpoint[j] = i;
}
```

Numero di trapezoidi di una triangolazione di n punti

Il numero totale di trapezoidi è dato dal numero di trapezoidi dentro la triangolazione (il doppio dei triangoli) più i trapezoidi formati tra il bounding box ed il convex hull della triangolazione (il numero di vertici + 2).
quindi il numero di trapezoidi è:

$$t = 2(2n - 2 - k) + k + 2 = 4n - k - 2$$

Posso anche esprimere t in funzione di n e del numero totale di segmenti s della triangolazione di n punti con m triangoli.
Il numero totale di spigoli di una triangolazione di punti con m triangoli è:

$$s = (3m + k) / 2$$

Il numero totale di triangoli è:

$$m = 2n - 2 - k$$

Ricavo i punti del Convex hull ponendo a sistema m ed s. Ottengo

$$k = 3n - s - 3$$

Sostituisco k nella formula precedente:

$$t = 4n - k - 2 = 4n - 3n + s + 3 - 2 = n + s + 1$$

quindi:

$$t = n + s + 1$$

dove:

t = numero di trapezoidi
n = numero totale di punti
s = numero di segmenti

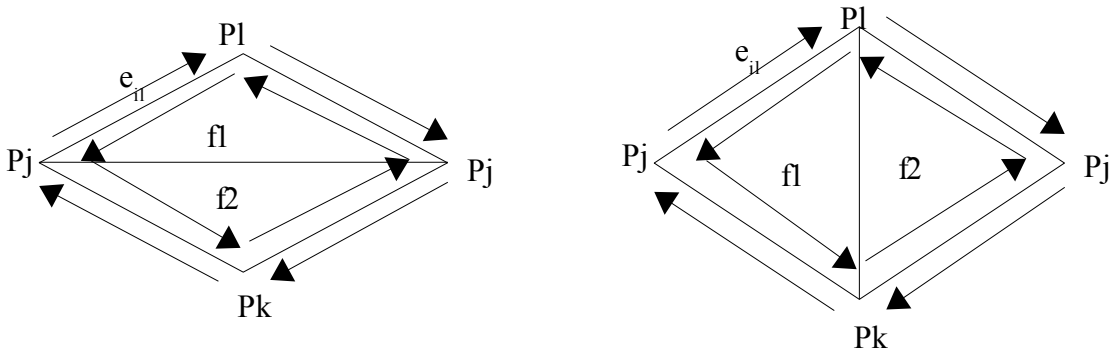
DCEL

Vertex	Coords	IncidentEdge
P_i		e_{ij}
P_k		e_{kj}
P_j		e_{il}
P_l		e_{li}

Face	OuterComponent	InnerComponent
f_∞	nil	e_{il}
f_1	e_{jl}	nil
f_2	e_{ji}	l

Half-edge	Origin	Twin	IncidentFace	Next	Prev
e_{ik}	P_i	e_{ki}	f_2	e_{kj}	e_{ji}
e_{kj}	P_k	e_{jk}	f_2	e_{ji}	e_{ik}
e_{ji}	P_j	e_{ij}	f_2	e_{ik}	e_{kj}
e_{ij}	P_i	e_{ji}	f_1	e_{jl}	e_{li}
e_{jl}	P_j	e_{lj}	f_1	e_{li}	e_{ij}
e_{li}	P_l	e_{il}	f_1	e_{ij}	e_{jl}
e_{il}	P_i	e_{li}	f_∞	e_{lj}	e_{ki}
e_{lj}	P_l	e_{jl}	f_∞	e_{jk}	e_{il}
e_{jk}	P_j	e_{kj}	f_∞	e_{ki}	e_{lj}
e_{ki}	P_k	e_{ik}	f_∞	e_{il}	e_{jk}

Edge Flip



EDGE FLIP (TriangulationD, Halfedge e_{ij})

Input:

DCEL D che descrive una triangolazione

half-edge e_{ij} che insiste sull'edge da flippare

//Ricavo i vertici della diagonale corrente e_{ij}

P_i = origin(e_{ij});

P_j = origin(twin((e_{ij}));

//Ricavo i vertici della nuova diagonale

P_1 = origin(prev((e_{ij}));

P_j = origin(prev(twin(e_{ij})));

//Ricavo le facce che incidono sulla diagonale

f_1 = IncidentFace(e_{ij});

f_2 = IncidentFace(twin((e_{ij}));

//Ricavo i nuovi next e prev

N_1 = prev((e_{ij});

P_1 = next(twin(e_{ij}));

N_2 = prev(twin(e_{ij}));

P_2 = next(e_{ij});

//Aggiorno la lista dei vertici

if IncidentEdge(P_i) = e_{ij}

then P_i.IncidentEdge = P_1

if IncidentEdge(P_j) = twin(e_{ij})

then P_j.IncidentEdge = P_2

//Aggiorno la lista delle facce

if OuterComponent(f_1) = e_{ij}

then f1.OuterComponent = N1

if OuterComponent(f_2) = twin(e_{ij})

then f2.OuterComponent = N2

//Aggiorno la lista degli half-edge

next(twin(e_{ij})).prev = N_1

prev(e_{ij})).next = P_1

next(e_{ij}).prev = N_2

prev(twin(e_{ij})).next = P_2

//Sostituisco e_{ij} con:

e_{kl}	P_k	e_{lk}	f_1	N_1	P_1

//Sostituisco twin(e_{ij}) con:

e_{kl}	P_l	e_{lk}	f_2	N_2	P_2

Return D;