Attenzione: il seguente documento è ancora in fase di stesura, pertanto presenta sezioni abbozzate, incorrette ed incomplete.

Tesi (titolo da definire)

Gianluca Mondini

21 settembre 2018

Indice

1 Obiettivo

Implementazione dell'algoritmo di Lloyd per l'equidistribuzione di uno sciame di droni su un'area specifica.

È prevista la realizzazione di un modulo nell'ambiente *OpenModelica* che permetta di simulare il suddetto algoritmo; più avanti verranno illustrate parti di codice e verranno fornite informazioni relative alla sua implementazione.

2 Parole chiave

Prima di procedere con la trattazione, è necessario indicare alcune parole chiave presenti in questo documento e nel codice dell'implementazione

Drone Velivolo che, nel nostro caso di interesse, è rappresentato come un punto bidimensionale

Area Poligono convesso che delimita l'algoritmo;

Cella Porzione di spazio alla quale appartiene un drone; la cella relativa ad un drone è l'insieme di punti dell'area per i quali, tra tutti i vari droni, quello appartenente alla cella è il più vicino

3 Il drone e sue componenti

Possiamo schematizzare, ai fini di questa trattazione, un *drone* mediante 3 componenti: il *centro fisico*, il *controllo di volo* e il *controllo della traiettoria*[1]

Inserire qui uno schemino che illustri i tre componenti del drone

3.1 Centro fisico

Il centro fisico è un astrazione che rappresenta il rapporto del drone con il mondo esterno. Vengono quindi qui considerati parametri la massa, la velocità, l'accelerazione, la posizione, la rotazione, la velocità di rotazione dei motori e la potenza che viene fornita a questi.

3.2 Controllo di volo

Il centro di volo considera il drone come un velivolo, ed utilizzando i dati forniti dal centro fisico e dal controllo della traiettoria permette il controllo della direzione, della quota e della velocità.

3.3 Controllo della traiettoria

Il controllo della traiettoria ha la funzione di comunicare al centro di volo la posizione da raggiungere.

Viene qui utilizza la funzione TargetPos, la cui ideazione ed implementazione viene illustrata in seguito.

[1] da rivedere i nomi dei 3 componenti

4 Cenni teorici sull'algoritmo di Lloyd

4.1 Introduzione

L'algoritmo di Lloyd, conosciuto anche con il nome di *iterazione di Voronoi*, è un algoritmo che permette di suddividere un'area in celle convesse uniformemente dimensionate.

L'algoritmo di Lloyd è implementato mediante iterazioni continue dell'algoritmo di Voronoi.

4.2 Descrizione

L'algoritmo esegue ripetutamente i seguenti *step*:

1. Viene generato il diagramma di Voronoi
2. Per ogni cella trovata, viene determinato il *baricentro*
3. Ogni punto viene spostato in corrispondenza del *baricentro* della propria *cella di Voronoi*

4.3 Esempio di applicazione dell'algoritmo

Viene qui presentata l'applicazione dell'algoritmo di Lloyd ad un'area quadrata nella quale sono presenti 5 partizioni.

Le croci rappresentano i *baricentri* delle varie partizioni.

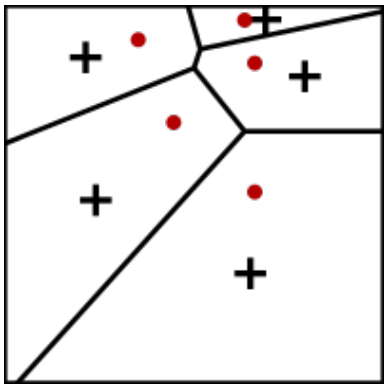


Figura 1: Iterazione n. 1; i droni, rappresentati dai punti rossi, non si trovano in corrispondenza del centro di massa della cella di appartenenza

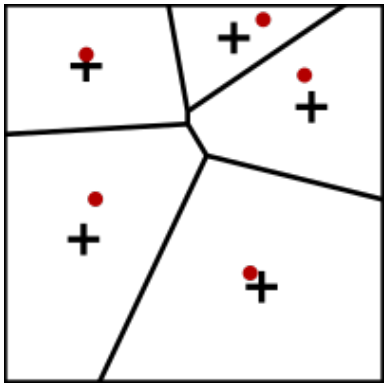


Figura 2: II iterazione

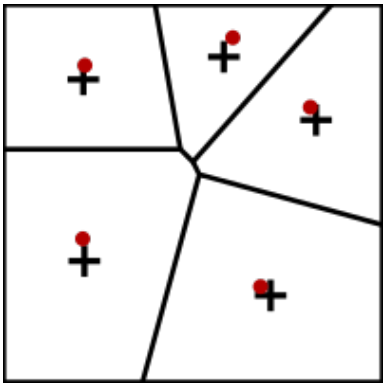


Figura 3: III iterazione

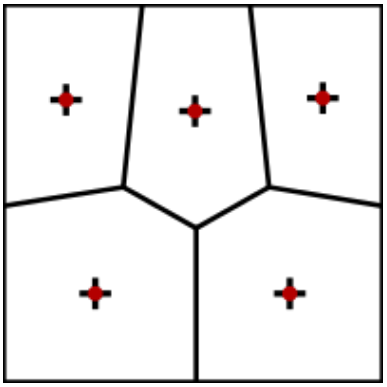


Figura 4: IV iterazione

4.4 Convergenza dell'algoritmo

Intuitivamente, si può dire che l'algoritmo converga in quanto i punti che si trovano a minor distanza tra loro tendono a compiere un movimento più alto, mentre i punti che si trovano a distanze elevate tendono a muoversi meno.

5 Implementazione in *OpenModelica*

5.1 Introduzione

L'implementazione dell'algoritmo di Lloyd richiede una funzione che effettui la tassellazione di Voronoi, la quale a sua volta utilizza una serie di funzioni geometriche che operino nello spazio bidimensionale.

Nell'implementazione che verrà illustrata successivamente, si è scelto di implementare in primo luogo l'algoritmo utilizzando il linguaggio *Python* per avere a disposizione una maggior quantità di strumenti di debugging e testing; successivamente è stato effettuato il porting del codice in *OpenModelica*

5.2 Diagramma delle dipendenze

Viene qui illustrato un diagramma contenenti le varie funzioni dell'implementazione *OpenModelica*. Le frecce $\longrightarrow$ rappresentano una dipendenza; ad esempio $A \longrightarrow B$ indica che A effettua una chiamata a B durante la sua esecuzione, e di conseguenza A *dipende da* B.

Sono state omesse le funzioni di supporto al testing e al debugging in quanto non strettamente legate all'implementazione di per sè.

In alto è presente la funzione TargetPos, la quale dipende da CenterOfMass, VoronoiCell e EdgesToVertices.

In basso troviamo CompareReal, che dipende esclusivamente da funzioni integrate in *OpenModelica*.

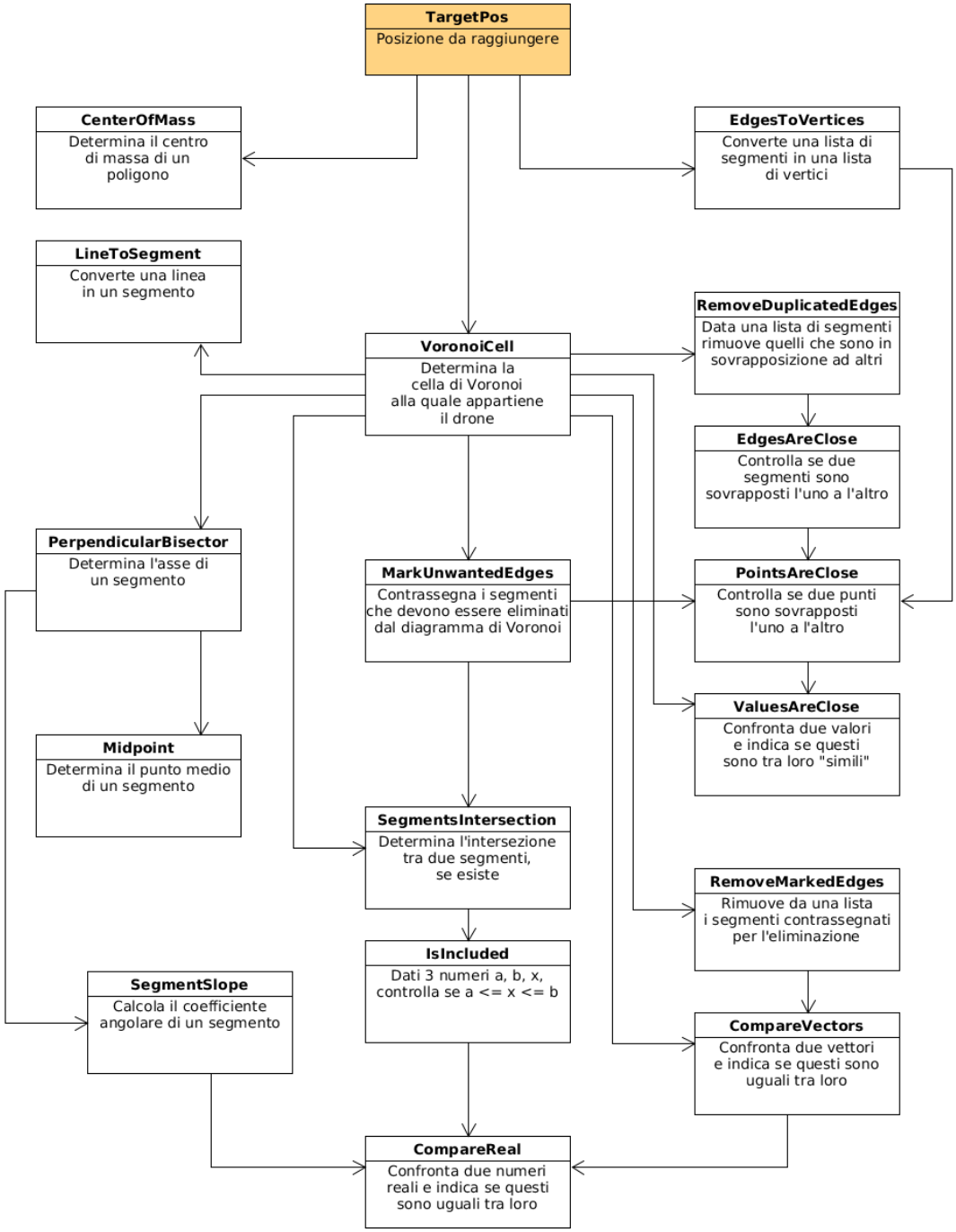


Figura 5: Diagramma delle dipendenze delle varie funzioni

5.3 Funzioni e strutture dati geometriche

Vengono qui descritte alcune delle funzioni implementate, seguendo l'approccio *bottom-up*, ovvero viene seguito in ordine inverso il grafo delle dipendenze

5.3.1 Scelte implementative

Come si vedrà in seguito, anzichè definire *tipi* personalizzati si è deciso di ricorrere alle strutture dati già presenti in *OpenModelica*.

Ad esempio, un punto dello spazio può essere rappresentato sia come *record* tramite

```
record Point
        Real x, y;
end Point;
```

sia come vettore

```
Real [2] point;
```

La scelta è ricade su quest'ultima implementazione in quanto è possibile utilizzare una serie di funzioni primitive (principalmente per lavorare su liste) già definite in *OpenModelica* che avrebbero richiesto altrimenti richiesto l'implementazione manuale tramite un linguaggio di più basso livello.

I punti a sfavore dell'utilizzo di un vettore al posto di una struttura dati personalizzata sono:

- L'impossibilità di accedere ai membri tramite il loro nome bensì tramite l'indice legato alla loro posizione. Ad esempio, per accedere alla coordinata y di un punto non è possibile utilizzare `point.y` ma `point[2]`, perdendo quindi di leggibilità
- L'impossibilità di aggiungere ulteriori campi contenenti informazioni. A questo riguardo si veda la sezione 5.5

5.3.2 Rappresentazione del punto geometrico

Per poter indicare un punto nello spazio bidimensionale, è necessario definire una struttura dati rappresentante una tupla x, y. A questo fine viene utilizzato un vettore bidimensionale definito tramite

```
Real [2] point;
```

Nel caso in cui debba essere dichiarato un vettore di punti, si dichiara una matrice tramite

```
Real [:, 2] some_points;
```

Il carattere `:` posto in prima posizione tra le parentesi quadre indica che la prima dimensione della matrice `some_points` non è conosciuta a priori.

5.3.3 Verifica della sovrapposizione di due punti

Può verificarsi che, in seguito a ..., ci si trovi nella situazione in cui due punti sovrapposti non superino il test di equalità. Ad esempio:

$$P_1 = (4.0, 2.0)$$

$$P_2 = (4.00001, 1.999999)$$

è chiaro come i due punti siano in realtà sovrapposti, ed è quindi necessario definire una funzione che si occupi di verificarlo

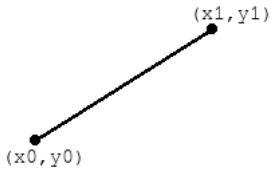
```
function PointsAreClose
    input Real [2] p1;
    input Real [2] p2;
    output Boolean are_close;
algorithm
    if ValuesAreClose(p1[1], p2[1]) then
        if ValuesAreClose(p1[2], p2[2]) then
            are_close := true;
        else
            are_close := false;
        end if;
    else
        are_close := false;
    end if;
end PointsAreClose;
```

5.3.4 Rappresentazione del segmento geometrico

Per rappresentare un segmento, si considera un vettore di 4 elementi, nella forma

$$(x_0, y_0, x_1, y_1)$$

dove (x_0, y_0) è il punto iniziale del segmento e (x_1, y_1) è il punto finale.



Questo si traduce, in ambiente *OpenModelica*, in:

```
Real [4] edge;

edge[1] := x_1;
edge[2] := y_1;
edge[3] := x_2;
edge[4] := y_2;
```

5.3.5 Verifica della sovrapposizione di due segmenti

La verifica della sovrapposizione di due segmenti viene effettuata in modo analogo a quella della sovrapposizione di due punti, e fa uso proprio di quest'ultima funzione.

Formalmente, due segmenti sono considerati sovrapposti se il punto iniziale del primo si sovrappone al punto iniziale del secondo e contemporaneamente il punto finale del primo si sovrappone al punto finale del secondo.

```
function EdgesAreClose
    input Real [4] e1, e2;
    output Boolean are_close;
algorithm
    are_close := PointsAreClose({e1[1], e1[2]}, {e2[1], e2[2]}) and
        PointsAreClose({e1[3], e1[4]}, {e2[3], e2[4]});
end EdgesAreClose;
```

5.3.6 Linea

Una *linea* è matematicamente identificata da una tupla di 3 elementi a, b e c tali che

$$ax + by + c = 0$$

In *OpenModelica*, si rappresenterà quindi come un vettore di 3 elementi

```
Real [3] line;

line[1] := a;
line[2] := b;
line[3] := c;
```

5.3.7 Conversione da linea a segmento

Può essere necessario convertire una linea, priva di un punto di inizio e fine, in un segmento, ben delimitato. Per fare ciò è stata implementata la funzione

$$\text{LineToSegment} : R[3] \longrightarrow R[4]$$

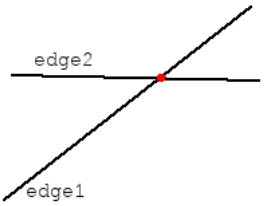
```
function LineToSegment
    input Real [3] line;
    output Real [4] segment;
protected
    parameter Real big = 10000;
    Real a, b, c;
    Real [2] p1, p2;
    Real m, q;
    Real y1, y2;
algorithm
    a := line[1];
    b := line[2];
    c := line[3];

    if b == 0 and (not (a == 0)) then // Retta verticale
        p1 := {-c, big};
        p2 := {-c, -big};
        segment := {p1[1], p1[2], p2[1], p2[2]};
        return;
    else
        m := -a/b;
        q := -c/b;
        y1 := m * (-big) + q;
        y2 := m * (+big) + q;
        segment := {-big, y1, big, y2};
    end if;
end LineToSegment;
```

5.3.8 Intersezione tra segmenti

Una delle funzioni principalmente utilizzate dall'algoritmo di tassellazione di Voronoi è quella responsabile di determinare l'eventuale punto di intersezione tra due segmenti.

È necessario notare che, avendo a che fare con segmenti di lunghezza finita e non con delle rette, è possibile che due segmenti non si intersechino tra loro pur non essendo paralleli.



L'algoritmo implementato è il seguente [2]

```
function SegmentsIntersection
    input Real [4] edge1, edge2;
    output Boolean valid;
    output Real [2] intersection;
protected
    Real x1, x2, y1, y2, dx1, dy1, x, y, xB, yB, dx, dy, DET, DETinv, r, s, xi,
         yi;
    parameter Real DET_TOLERANCE = 0.00000001;
algorithm
    x1 := edge1[1];
```

[2] Il codice è un adattamento di https://www.cs.hmc.edu/ACM/lectures/intersections.html

```
10      y1 := edge1[2];
11      x2 := edge1[3];
12      y2 := edge1[4];
13
14      x := edge2[1];
15      y := edge2[2];
16      xB := edge2[3];
17      yB := edge2[4];
18
19      dx1 := x2 - x1;
20      dy1 := y2 - y1;
21
22      dx := xB - x;
23      dy := yB - y;
24
25      DET := ((-dx1 * dy) + (dy1 * dx));
26
27      if abs(DET) < DET_TOLERANCE then
28          valid := false;
29          return;
30      end if;
31
32      DETinv := 1.0/DET;
33
34      r := DETinv * (-dy  * (x-x1) +  dx * (y-y1));
35      s := DETinv * (-dy1 * (x-x1) + dx1 * (y-y1));
36      xi := (x1 + r*dx1 + x + s*dx)/2.0;
37      yi := (y1 + r*dy1 + y + s*dy)/2.0;
38
39      if (IsIncluded(0, 1, r)) and (IsIncluded(0, 1, s)) then
40          intersection := {xi, yi};
41          valid := true;
42          return;
43      else
44          valid := false;
45          return;
46      end if;
47  end SegmentsIntersection;
```

Oltre a restituire un vettore bidimensionale `intersection` viene anche restituito un booleano `valid`, che sta ad indicare se il valore contenuto in `intersection` è valido o meno.

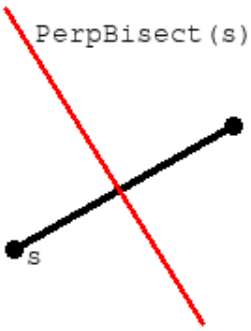
5.3.9 Asse di un segmento

La funzione VoronoiCell fa ampio uso della funzione PerpendicularBisector, che restituisce l'asse del segmento passato come argomento.

$$\text{PerpendicularBisector} : \text{Segmento} \longrightarrow \text{Linea}$$

Vengono discriminati i 3 casi in cui:

- Il segmento sia verticale; in tal caso l'asse sarà orizzontale;
- Il segmento sia orizzontale; in tal caso l'asse sarà verticale;
- Il segmento non sia nè verticale nè orizzontale: caso più frequente




```
function PerpendicularBisector
    input Real [4] edge;
    output Real [3] perp_bisect;
protected
    Boolean vertical;
    Real [2] p;
    Real a, b, c, neg_c, m1, m2, q;
algorithm
    p := Midpoint(edge);
    (m1, vertical) := SegmentSlope(edge);
    if vertical then
        neg_c := (edge[2] + edge[4])/2;
        perp_bisect := {0, 1, -neg_c};
        return;
    elseif m1 == 0 then
        a := 1;
        b := 0;
        c := -(edge[1] + edge[3])/2;
        perp_bisect := {a, b, c};
        return;
    else
        m2 := -1/m1;
        q := - m2 * p[1] + p[2];
        perp_bisect := {-m2, 1, -q};
        return;
    end if;
end PerpendicularBisector;
```

5.4 Centro di massa

Al fine di implementare l'algoritmo di Lloyd è necessario definire una funzione che, presa in ingresso una lista di vertici, ne calcoli il centro di massa.

Dato un poligono di n vertici $(x_0, y_0), (x_1, y_1), ..(x_{n-1}, y_{n-1})$ il centro di massa ha coordinate (C_X, C_Y) definite come

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

dove A è l'area del poligono *con segno*

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

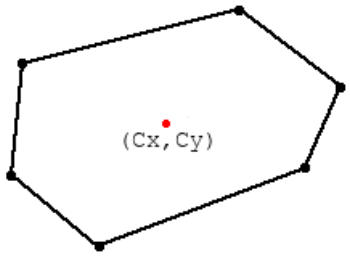


Figura 6: Il punto p viene restituito dalla chiamata a CenterOfMass che ha come argomento la lista delle coordinate dei vari vertici del poligono

Ne deriva quindi la seguente codifica:

```
function CenterOfMass
    input Real [:, 2] points;
    output Real [2] out;
protected
    Real [:, 2] vertices, v_local;
    Real x_cent, y_cent, area, factor;
algorithm
    // Adattato da https://stackoverflow.com/a/46937541
    vertices := points;

    x_cent := 0;
    y_cent := 0;
    area := 0;

    v_local := cat(1, vertices, {vertices[1]});

    for i in 1:(size(v_local, 1) - 1) loop
        factor := v_local[i,1] * v_local[i+1, 2] - v_local[i+1,1] * v_local[i,2
            ];
        area := area + factor;
        x_cent := x_cent + (v_local[i, 1] + v_local[i+1,1]) * factor;
        y_cent := y_cent + (v_local[i, 2] + v_local[i+1, 2]) * factor;
    end for;

    area := area / 2.0;
    x_cent := x_cent / (area * 6);
    y_cent := y_cent / (area * 6);

    out := {x_cent, y_cent};
end CenterOfMass;
```

5.5 Ricerca ed eliminazione dei segmenti in eccesso

Durante l'esecuzione dell'algoritmo di Voronoi si presenta la necessità di rimuovere dei segmenti che non fanno più parte del diagramma. Si presenta il problema di come effettuare questa operazione senza arrecare danno all'iterazione in corso proprio sulla lista dalla quale vanno rimossi gli elementi.

Inoltre, come precedentemente detto, non è possibile aggiungere un campo ad ogni segmento indicando l'effettiva necessità di eliminazione.

La scelta cade quindi sull'assegnare il valore -1 ad ogni componente del segmento, considerandolo quindi un segmento nullo, da eliminare.

```
edge := {-1, -1, -1, -1};
```

Successivamente, utilizzando la funzione `CompareVector` si controlla se il segmento è contrassegnato per l'eliminazione

```
if CompareVector(edge, {-1, -1, -1, -1}) then
```

```
        // Considera 'edge' come un segmento da eliminare
else
        // Considera 'edge' come un segmento valido
end if;
```

La funzione `MarkUnwantedEdges` contrassegna tutti i segmenti che devono essere rimossi dalla lista di segmenti. In particolare, se un segmento si trova dietro ad un altro quello "nascosto" deve essere eliminato in quanto non può più far parte dell'insieme.

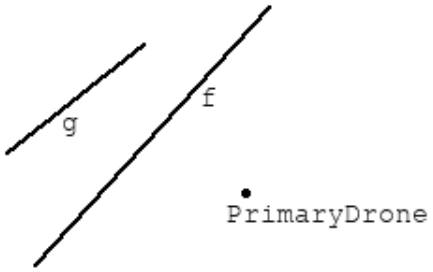


Figura 7: Esempio nel quale il segmento g deve essere rimosso, in quanto il segmento f si pone tra esso e il *PrimaryDrone*. La funzione MarkUnwantedEdges contrassegnerà g per l'eliminazione, mentre RemoveMarkedEdges lo cancellerà definitivamente.

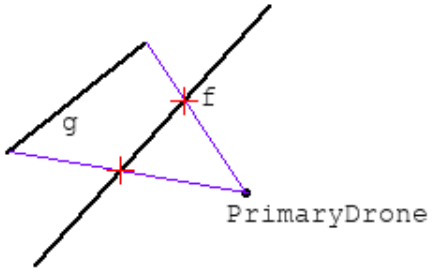


Figura 8: I segmenti di unione tra PrimaryDrone e gli estremi di g collidono con il segmento f; questo fa sì che g venga contrassegnato per l'eliminazione.

```
function MarkUnwantedEdges
    input  Real [:,4] edges;
    input  Real [2]   primary_drone;
    output Real [:,4] marked_edges;
protected
    Real [0,4] empty_marked_edges;
    Real [2] point, intersection;
    Real [4] inner_edge, join_edge;
    Boolean valid;
algorithm
    marked_edges := edges;
    for outer_i in 1:size(edges, 1) loop
        for point_index in 1:2 loop
            if point_index == 1 then
                point := {edges[outer_i,1], edges[outer_i,2]};
            elseif point_index == 2 then
                point := {edges[outer_i,3], edges[outer_i,4]};
            end if;
```

```
            for inner_edge_index in 1:size(edges, 1) loop
                inner_edge := edges[inner_edge_index];
                join_edge := {primary_drone[1], primary_drone[2], point[1],
                    point[2]};
                (valid, intersection) := SegmentsIntersection(inner_edge,
                    join_edge);
                if valid and (not PointsAreClose(intersection, point)) then
                    marked_edges[outer_i] := {-1, -1, -1, -1};
                else
                end if;
            end for;
        end for;
    end for;
end MarkUnwantedEdges;
```

La funzione `RemoveMarkedEdges` restituisce una lista di segmenti dalla quale sono stati rimossi tutti i segmenti contrassegnati per l'eliminazione.

```
function RemoveMarkedEdges
    input Real[:,4] edges;
    output Real [:,4] clean_edges;
protected
    Real [0,4] empty_clean_edges;
algorithm
    clean_edges := empty_clean_edges;
    for i in 1:size(edges, 1) loop
        if not CompareVectors(edges[i], {-1, -1, -1, -1}) then
            clean_edges := cat(1, clean_edges, {edges[i]});
        end if;
    end for;
end RemoveMarkedEdges;
```

5.6 La funzione VoronoiCell

Come precedentemente detto, l'algoritmo di Lloyd è implementato tramite iterazioni successive dell'algoritmo di Voronoi.

La funzione VoronoiCell restituisce esclusivamente la cella di appartenenza di un drone specificato, anziché la tassellazione dell'intera area; considerato che l'algoritmo è distribuito ogni drone necessita di conoscere esclusivamente la propria posizione *target*, e soltanto la posizione attuale dei restanti droni.

L'idea è quindi quella di implementare la seguente funzione:

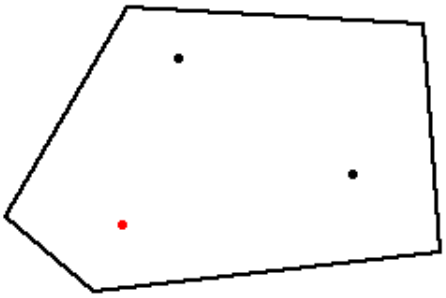
$$\text{VoronoiCell} : (\text{drone stesso}, \text{altri droni}, \text{bordi}) \longrightarrow \text{cella drone stesso}$$

I passaggi fondamentali dell'algoritmo implementato sono i seguenti:

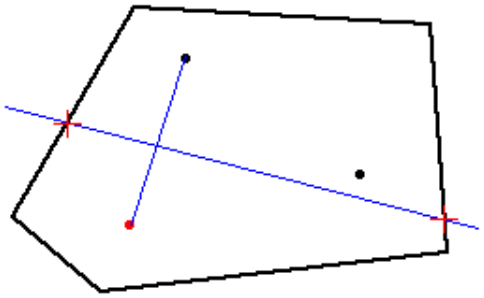
1. Per ogni drone contenuto in other drones (ovvero la lista degli altri droni)
   (a) viene creato un segmento union edge che unisce il drone stesso (primary drone) con drone
   (b) viene determinato l'asse di union edge, che prende il nome di perp bisect
   (c) viene inizializzata una lista vuota intersections che andrà a contenere i punti di intersezione trovati
   (d) Per ogni bordo edge contenuto in edges
      i. viene cercato l'eventuale punto di intersezione tra perp bisect ed edge
      ii. se il punto di intersezione esiste
         A. contrassegno edge per la cancellazione, in quanto verrà sostituito da un nuovo bordo
         B. determino quale estremo di edge conservare per la creazione del nuovo bordo
         C. aggiunto il punto di intersezione trovato alla lista intersections
   (e) se intersections contiene 2 punti, creo un nuovo bordo che abbia come estremi i 2 punti
2. tolgo da edges i bordi duplicati e quelli contrassegnati per l'eliminazione

5.6.1 Esempio grafico di applicazione dell'algoritmo di Voronoi

Vediamo adesso l'applicazione dell'algoritmo di Voronoi alla seguente situazione: l'area è delimitata da un poligono a 5 lati; in rosso è visibile quello che viene chiamato PrimaryDrone, ovvero il drone che effettua la chiamata a VoronoiCell e per il quale si intende calcolare la cella associata; i restanti 2 puntini neri rappresentano gli altri due droni.

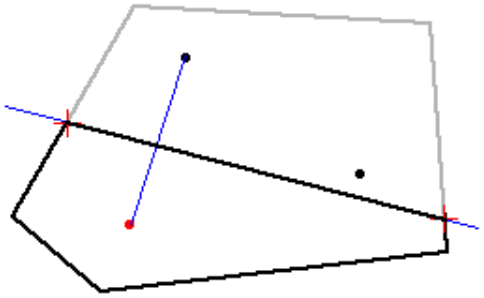


Si determina quindi il segmento che unisce PrimaryDrone con uno degli altri droni, in questo caso quello più in alto; si traccia l'asse del segmento trovato e si determinano i punti di intersezione con i bordi dell'area, che nella figura sono contrassegnati da croci rosse.



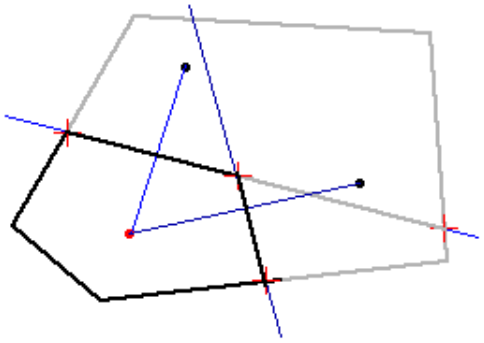
Si crea un nuovo bordo che unisce i due punti di intersezione. Tutti i segmenti che si trovino "dietro" (si veda la sezione 5.5) al nuovo segmento vengono contrassegnati per l'eliminazione.

Si noti inoltre che i due bordi con i quali l'asse intersecava sono stati sostituiti da due nuovi bordi, di lunghezza inferiore.



Si ripete il procedimento anche per l'altro drone, ed essendo in questo caso i droni totali in numero pari a 3 si conclude l'algoritmo.

Il poligono contornato in nero è quindi la cella di Voronoi associata al PrimaryDrone



Il codice che ne deriva è il seguente:

```
function VoronoiCell
    input  Real    [:, 4]  input_edges;
    input  Real    [2]     primary_drone;
    input  Real    [:, 2]  other_drones;
    output Real    [:, 4]  output_edges;
protected
    Real    [:, 4]  edges;
    Real    [2]     drone, point, intersect, p1, p2, int1, int2, keep;
    Real    [4]     union_edge, edge_p1_primary_drone, edge_p2_primary_drone,
        new_edge;
    Real    [3]     perp_bisect;
    Real    [:,2]   intersections;    Real [0,2] empty_intersections;
    Real    [:,4]   new_edges;        Real [0,4] empty_new_edges;
    Boolean have_intersection, int1_valid, int2_valid, add_to_intersections;
algorithm
    edges := input_edges;
    for other_drones_index in 1:size(other_drones, 1) loop drone :=
        other_drones[other_drones_index];
        union_edge := {primary_drone[1], primary_drone[2], drone[1], drone[2]};
        perp_bisect := PerpendicularBisector(union_edge);
        intersections := empty_intersections;
        new_edges := empty_new_edges;
        edges := MarkUnwantedEdges(edges, primary_drone);
        for i in 1:size(edges, 1) loop
            if not CompareVectors(edges[i], {-1, -1, -1, -1}) then
                (have_intersection, intersect) := SegmentsIntersection(
                    LineToSegment(perp_bisect),
                    edges[i]
                );
                if have_intersection then
                    p1 := {edges[i, 1], edges[i, 2]};
                    p2 := {edges[i, 3], edges[i, 4]};
                    edges[i] := {-1, -1, -1, -1};
                    edge_p1_primary_drone := {p1[1], p1[2], primary_drone[1],
                        primary_drone[2]};
                    edge_p2_primary_drone := {p2[1], p2[2], primary_drone[1],
                        primary_drone[2]};
                    (int1_valid, int1) := SegmentsIntersection(
                        LineToSegment(perp_bisect),
                        edge_p1_primary_drone
                    );
                    (int2_valid, int2) := SegmentsIntersection(
                        LineToSegment(perp_bisect),
                        edge_p2_primary_drone
                    );
                    assert(int1_valid or int2_valid, "non viene creata un
                        intersezione ne con p1 ne con p2. p1 = " +
                        VectorToString(p1) + ", p2 = " + VectorToString(p2) + "
                        , primary_drone = " + VectorToString(primary_drone));
                    if int1_valid and not int2_valid then
                        keep := p2;
                    elseif not int1_valid and int2_valid then
                        keep := p1;
                    elseif int1_valid and int2_valid then
```

```
                    else
                        assert(false, "errore logico");
                    end if;
                    new_edge := {intersect[1], intersect[2], keep[1], keep[2]};
                    new_edges := cat(1, new_edges, {new_edge});
                    add_to_intersections := true;
                    for intersections_index in 1:size(intersections, 1) loop
                        if ValuesAreClose(intersections[intersections_index,
                            1], intersect[1]) and ValuesAreClose(intersections[
                            intersections_index, 2], intersect[2]) then
                            add_to_intersections := false;
                        end if;
                    end for;
                    if add_to_intersections then
                        intersections := cat(1, intersections, {intersect});
                    end if;
                end if;
                edges := MarkUnwantedEdges(edges, primary_drone);
            end if;
        end for;
        edges := cat(1, edges, new_edges);

        if size(intersections, 1) == 2 then
            edges := cat(1, edges, {{intersections[1,1], intersections[1,2],
                intersections[2,1], intersections[2,2]}});
        end if;

    end for;

    assert(size(edges, 1) > 1, "edges deve contenere almeno un elemento");

    edges := MarkUnwantedEdges(edges, primary_drone);
    edges := RemoveMarkedEdges(edges);
    edges := RemoveDuplicatedEdges(edges);

    output_edges := edges;
end VoronoiCell;
```

5.7 La funzione TargetPos

La funzione TargetPos effettua una singola iterazione dell'algoritmo di Lloyd, ovvero:

1. Determina la cella di Voronoi relativa al drone stesso
2. Calcola il centro di massa della cella determinata al punto 1

```
function TargetPos

    input   Real  [2]    self_position;
    input   Real  [:,2]  other_drones_positions;
    input   Real  [:,4]  area_boundaries;

    output  Real  [2]    target_position;

algorithm
    target_position := CenterOfMass(
        EdgesToVertices(
            VoronoiCell(
                area_boundaries,
                self_position,
                other_drones_positions
            )
        )
    );
end TargetPos;
```

5.8 Testing del codice

Attualmente *OpenModelica* non fornisce un meccanismo di testing solido e standardizzato. Per questo motivo è stato necessario realizzare un sistema che automatizzasse la verifica della correttezza delle funzioni implementate.

A questo proposito si è deciso di ricorrere ad uno script *Python* che:

1. Ottenga la lista delle classi implementate
2. Ottenga la lista dei test da effettuare
3. Compili le classi
4. Crei un file contenente le istruzioni per eseguire tutti i test
5. Esegua, uno dopo l'altro, i vari test
6. Restituisca a schermo un output indicante l'esito dei vari test

```
#!/bin/env python3

import re
import subprocess
import os
import sys

CLASSES_FILENAMES = list(
    filter(
        lambda name: name.endswith(".mo"),
        os.listdir("classes")
    )
)
CLASSES_FILENAMES = list(
    map(
        lambda name: "classes/" + name,
        CLASSES_FILENAMES
    )
)

TEST_FILENAMES = list(
    filter(
        lambda name: name.endswith(".mo"),
        os.listdir("test")
    )
)
TEST_FILENAMES = list(
    map(
        lambda name: "test/" + name,
        TEST_FILENAMES
    )
)

FILENAMES = CLASSES_FILENAMES + TEST_FILENAMES

class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'

os.system("clear")

forced_tests = list(
    filter(
        lambda s: s.startswith("test_"),
```

```
        sys.argv
    )
)

content = ""
for filename in FILENAMES:
    with open(filename) as f:
        content += f.read()

identifiers = re.findall(r"test_\w+", content)
identifiers = list(set(identifiers))

with open("tmp_simulate.mos", mode="w+") as f:
    for filename in FILENAMES:
        f.write(f'loadFile("{filename}"); getErrorString();\n')
    f.write('cd("/tmp");\n')
    if forced_tests:
        for t in forced_tests:
            f.write(f"simulate({t}); getErrorString();\n")
            f.write('print
                ("--------------------------------------------------------------------")
                ;\n')
    else:
        for i in identifiers:
            f.write(f"simulate({i}); getErrorString();\n")
            f.write('print
                ("--------------------------------------------------------------------")
                ;\n')

result = subprocess.run(['omc', './tmp_simulate.mos'], stdout=subprocess.PIPE)
result = result.stdout.decode()

if "--raw" in sys.argv:
    print(result)
else:
    for func in [
        lambda line: not line.startswith("    time"),
        lambda line: "record SimulationResult" not in line,
        lambda line: "end SimulationResult" not in line,
        lambda line: line != '""',
        lambda line: line != 'true',
    ]:
        result = "\n".join(filter(func, result.split("\n")))
    result = result.replace("The simulation finished successfully.", bcolors.
        OKGREEN + "The simulation finished successfully!" + bcolors.ENDC)
    result = result.replace("assert", bcolors.FAIL + "assert" + bcolors.ENDC)
    for i in identifiers:
        result = result.replace(i, bcolors.BOLD + i + bcolors.ENDC)
    print(result)
```

Bibliografia

- https://en.wikipedia.org/wiki/Lloyd%27s_algorithm
- github.com/zapateocallisto
- https://en.wikipedia.org/wiki/Centroid